.NET开发经典名著



ASP.NET MVC 5 编程实战(第3版)

——构建在桌面和移动设备运行同样精彩的Web应用



Professional



[美] Dino Esposito 著
潘丽臣 译

清华大学出版社

.NET 开发经典名著

ASP.NET MVC 5 编程实战

(第 3 版)

——构建在桌面和移动设备运行同样精彩的 Web 应用

[美] Dino Esposito 著

潘丽臣 译

清华大学出版社

北 京

图书在版编目(CIP)数据

ASP.NET MVC 5 编程实战(第 3 版)——构建在桌面和移动设备运行同样精彩的 Web 应用/(美)埃斯波西托(Esposito, D.) 著；潘丽臣 译. —北京： 清华大学出版社，2015
(.NET 开发经典名著)
书名原文：Programming Microsoft ASP.NET MVC, Third Edition
ISBN 978-7-302-39480-8

Ⅰ. ①A… Ⅱ. ①埃… ②潘… Ⅲ. ①网页制作工具—程序设计 Ⅳ. ①TP393.092

中国版本图书馆 CIP 数据核字(2015)第 036197 号

责任编辑：王　军　于　平
装帧设计：孔祥峰
责任校对：成凤进
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市金元印装有限公司
经　　销：全国新华书店
开　　本：185mm×230mm　　**印　　张：**30　　**字　　数：**653 千字
版　　次：2015 年 3 月第 1 版　　**印　　次：**2014 年　月第 1 次印刷
印　　数：1～4000
定　　价：59.80 元

产品编号：

译 者 序

凡是用过 ASP.NET MVC 的开发人员必定获益匪浅。ASP.NET MVC 具有耦合性低、重用性高、生命周期成本低、部署快、可维护性高、有利于软件工程化管理方面的优点。该架构模式对于大中型应用程序的架构和开发来说，优势尤为明显。

一直以来，国内大多数ASP.NET开发人员都限于用控件拼装Web表单进行Web应用程序的开发，这使得不少开发人员更关注用控件实现呈现和功能交互，而往往忽视了在应用程序总体架构和软件层次体系方面的考虑，也相应造成了Web应用程序良莠不齐的局面。随着ASP.NET MVC的逐步推广应用，相信这种情况会有所改善。

本书对 ASP.NET MVC 进行了全面阐述，读者通读本书后，会对 ASP.NET MVC 有更为详尽的了解。知易行难，希望有志于从事 Web 应用程序开发的读者能够从本书中获得启发并尝试用于实践，将书本内容转化成自己的知识。在此要特别强调的是，本书第III部分重点介绍了移动端开发方面的内容，也算是迎合了当前的时代潮流。管中窥豹，我们由此可以预见 ASP.NET 整体框架将愈发加大对移动端开发的支持，而 ASP.NET MVC 已经走在了前面，所以 ASP.NET 开发人员在转向移动端开发的同时也必然愈发感受到微软强大的支持后盾。

在此要特别感谢清华大学出版社的编辑们，他们在本书翻译过程中为译者提供的巨大帮助，没有其热情付出，本书将难以顺利付梓。本书全部章节由潘丽臣翻译，参与本次翻译活动的还有蒲成、杨晔、杨达辉、申成龙、杨帆、赵栋、王滨、李鹏、贠书谦、林超、陈世佳。在此一并表示感谢。

译者具有多年的 ASP.NET 开发经验，最近几年也一直在实际开发中应用 ASP.NET MVC 的先进理念和技术，深感 ASP.NET MVC 作为一种架构模式能为开发人员带来极大的便捷，其必将成为新的主流开发指导思想。所以，开发人员都应该尽早掌握 ASP.NET MVC，对于 ASP.NET 开发人员来说尤其如此。由于译者水平有限，难免会出现一些错误或翻译不准确的地方，如果有读者能够指出并勘正，译者将不胜感激。

译 者

作 者 简 介

Dino Esposito 是 e-tennis.net 网站的 CTO 和创始人之一，这是一个新创办的为专业网球和运动公司提供软件和 IT 服务的网站。Dino 仍然在进行大量的培训工作和写作，并著有多本 Web 开发和.NET 设计方面的书籍。他最新的两本著作是由微软出版社出版的 *Architecting Mobile Solutions for the Enterprise* 和 *Microsoft .NET: Architecting Applications for the Enterprise*。Dino 经常在行业会议(包括 DevConnections)以及像 Software Architect、DevWeek 和 BASTA 这样的欧洲著名活动中演讲。Dino 不仅是 JetBrains 公司在 Android 和 Kotlin 开发方面的技术专家，还是提供 WURFL 的 ScientiaMobile 数据库开发团队的成员。该数据库存储了大量移动设备性能，被多家大型组织使用，比如 Facebook。

你可以在 Twitter 上通过@despos 及其博客(http://software2cents.wordpress.com)来关注 Dino。

前　　言

首先要掌握事实，然后你可以随意歪曲它们。

——*Mark Twain*

ASP.NET 诞生于 20 世纪 90 年代末期各行各业正迅速探索互联网的时代。ASP.NET 的主要目的是为了让开发人员能够快速有效地构建应用程序，而无须处理如 HTTP、 HTML 和 JavaScript 等错综复杂的底层细节。这正是当时的社会环境所强烈要求的。ASP.NET 是微软推出来满足这项需求的，且大大超过了预期的程度。

十多年后的今天，ASP.NET 的发展显得有些滞后，很多人甚至开始质疑 Web 框架存在的必要性。这是一个了不起的时代，为我们提供了若干选项。其中就有 Web Forms 和 ASP.NET MVC 应用程序，还有更多 JavaScript 密集型客户端应用程序(单页面应用程序)，它们使用一个服务器端后台来为实际公开的一些页面提供基本布局和特设服务，比如捆绑。

奇妙的是，使用 Web Forms 模式，你仍可以编写功能性应用程序，尽管 ASP.NET MVC 能够更密切地服务于开发人员的当前需求。Web Forms 的最常见应用场景是，你要开发专注于呈现数据并使用优质第三方控件套装的应用程序。ASP.NET MVC 可用于处理其他所有方面，包括客户端单页面应用程序的框架搭建。

Web 应用程序的改变方式证明了，ASP.NET MVC 可能未能替代 ASP.NET Web Forms 在众多开发人员心目中的地位，但这却是正确的选择，ASP.NET MVC 足以成为任何一个需要实体后台的应用程序的理想 Web 平台，对于那些以多设备实用功能为目标的 Web 应用程序来说尤其如此。是的，这很可能意味着不到两年时间内的所有 Web 应用程序。

转换到 ASP.NET MVC，对于 ASP.NET 开发人员来说是相当自然的过程。

本书读者对象

这几年来，不少人读过我的一些书籍和文章。这些读者已经察觉到了，我并不擅长写作步骤详解类的参考型书籍，同样，我也不能对同一门课程在前后两次的教学中以相同的顺序介绍主题，或提供前后相同的例子。

此书并不适合绝对的初学者；但除此之外的其他人我觉得都可以阅读，包括那些对 ASP.NET MVC 还不甚了解的人。能力和专业水平越高的人，越难在本书中找到相关专业领域的附加值。然而，这本书得益于几年的现实实践，我相信其中一定有很多可能还会吸引专家的解决方案，尤其是涉及移动设备方面的。

如果你使用 ASP.NET MVC，我相信你一定会在此书中找到一些有价值的东西。

假定

这本书假定你对 ASP.NET 开发有基本的了解。

不适合阅读本书的人群

如果你需要的是 ASP.NET MVC 分步指南，那么本书算不上是一本理想的书籍。

本书结构

该书分为三个部分。第 I 部分："ASP.NET MVC 基础"，提供了对 ASP.NET 基础和其核心组件的简短概述。第 II 部分："ASP.NET MVC 软件设计"，着重介绍 Web 应用程序、特定设计模式和最佳实践的常见问题。最后，第III部分："移动客户端"，是有关 JavaScript 和移动界面的。

系统要求

需要安装以下软件以运行本书中所提供的示例：

- 以下的操作系统之一：Windows 8/8.1、Windows 7、Windows Vista with Service Pack 2 (除了简化版)、Windows XP with Service Pack 3(除了简化版)、Windows Server 2008 with Service Pack 2、Windows Server 2003 with Service Pack 2 以及 Windows Server 2003 R2。
- Microsoft Visual Studio 2013 的任意版本(如果你使用 Express Edition 产品，则可能需要多个下载)。

- Microsoft SQL Server 2012 Express Edition 或更高版本，以及 SQL Server Management Studio 2012 Express 或更高版本(与 Visual Studio 一起分发；Express Edition 需要单独下载)。

根据你的 Windows 配置，可能需要本地管理员权限才能安装或配置 Visual Studio 2013 和 SQL Server 2012 产品。

示例代码下载

该书的大多数章节都包含一些练习，你可以用交互的方式尝试正文中所学到的新材料。可以从以下网页下载所有的示例项目，包括它们实践前和实践后的格式：

```
http://aka.ms/programASP-NET_MVC/files
http://www.tupwk.com.cn/downpage
```

请按照说明下载 asp-net-mvc-examples.zip 文件。

安装代码示例

通过执行下列步骤，在你的计算机中安装代码示例，以便在你做本书中的练习时可以使用。

(1) 将对本书的网站上下载的 asp-net-mvc-examples.zip 文件进行解压(如有必要，指定一个特定的目录和路径来创建它)。

(2) 如果弹出提示，请查看所显示的最终用户许可协议。如果你接受这些条款，请选择 Accept 选项，然后单击 Next。

注意：
如果没有显示许可协议，你可以从下载 asp-net-mvc-examples.zip 文件的网页访问它。

使用示例代码

Setup.exe 程序所创建的文件夹包含每一章的一个子文件夹。反之，每一章可能包含额外的子文件夹。所有示例都被组织在一个单独的 Visual Studio 2013 解决方案中。你要打开 Visual Studio 2013 中的解决方案文件并导航到这些示例。

勘误及相关支持

我们尽一切努力确保此书及其同步内容的准确性。本书自出版以来所报告的一切错误都列在微软出版社网站上：

http://aka.ms/programASP-NET_MVC/errata

如果你发现了未列出的错误，可以通过相同的页面发送报告给我们。

如果你需要额外的支持，请发送电子邮件到 mspinput@microsoft.com 给微软出版社的支持部。

请注意，上述地址不提供微软软件的产品支持。

我们期待你的反馈

在微软出版社，你的满意才是我们的首要任务，你的反馈是我们最宝贵的财富。请告诉我们你对此书的看法：

http://aka.ms/tellpress

这项调查是短暂的，但我们认真阅读你的每一条意见和想法。提前感谢你的输入！

保持联系

让我们将交流继续下去！这里是我们的 Twitter 网址：http://twitter.com/MicrosoftPress。

目　　录

第 III 部分 移动客户端

第 I 部分

ASP.NET MVC 基础

第 1 章

ASP.NET MVC 控制器

人们总说时间会改变一切，但实际上你必须自己动手去改变一切。

——*Andy Warhol*

我认为从 Ajax 征服大众的那一天开始，ASP.NET Web Forms 就开始变得不堪胜任了。正如有些人所说，Ajax 已经成为一只射中 ASP.NET 之踵的另一支 Achilles 毒箭了。Ajax 使得对 HTML 和客户端代码拥有越来越多的控制权这一情形成为了事实存在。随着时间的推移，这引发了不同架构的形成，使 ASP.NET Web Forms 逐渐丧失了对任务的胜任能力。

由于可以适用于现有的 ASP.NET 运行时，MVC 模式产生了一个新的框架——ASP.NET MVC——它能让 Web 开发与开发人员的当前需求相匹配。

在 ASP.NET MVC 中，每个请求的结果最终都会执行某个操作——根本上来说也就是特定类上的方法。操作执行的结果会与一个视图模板一起传递给视图子系统。结果和模板随后会用于生成浏览器的最终响应。用户不需要将浏览器指向某个页面，他们只需要放置一个请求即可。难道这还不是很大的变化吗？

与 Web Forms 不同，ASP.NET MVC 是由连接在一起的各种代码层所组成，而不是交织在一起形成一个单一的整体块。鉴于此，可以很容易地以自定义组件替换任何层，用以提高解决方案的可维护性和可测试性。使用 ASP.NET MVC，可以获得对标记的完全控制，并能随意用你最喜欢的 JavaScript 框架来套用样式和注入脚本代码。

基于与 Web Forms 相同的运行时环境，ASP.NET MVC 推出了一个适合于 Web 应用程序的经典模型-视图-控制器(Model-View-Controller)模式，使 Web 应用程序的开发有了明显不同的体验。在这一章中，你将会了解控制器的结构与作用——ASP.NET MVC 应用程序的基础——以及请求被路由到控制器的方式。

尽管你有可能决定继续使用 Web Forms 来进行当前的 Web 开发，但 ASP.NET MVC 的确是更好的选择。虽然不需要投入大量的时间，但你需要确切地知道发生了什么以及 MVC 背后的原理。如果照此做，那么你的任何投入都将会比预期更早地得到回报。

注意：

本书基于 ASP.NET MVC 5。此版本的 ASP.NET MVC 兼容之前的版本。这意味着可以在同一台计算机上同时安装两个版本，在使用最新版本时不会影响你已经拥有的现成的 MVC 代码。

1.1　对输入请求进行路由

最初，整个 ASP.NET 平台都是围绕着服务物理页面请求的理念来开发的。事实证明大部分在 ASP.NET 应用程序中所使用的 URL 都由两部分组成：分别是包含了逻辑的物理网页路径，和填充在查询字符串中用于提供参数的一些数据。这种方法已经使用数年，现今仍然有效。然而，ASP.NET 运行时环境并不会限制你仅仅调用由特定位置和文件所确定的资源。通过编写一个专门的 HTTP 处理程序并将其绑定到一个 URL，就可以使用 ASP.NET 来执行代码以响应请求，而无须依赖物理文件。这只是 ASP.NET MVC 与 ASP.NET Web Forms 之间的一个主要区别。让我们大致了解一下如何用 HTTP 处理程序来模拟 ASP.NET MVC 行为。

注意：

在软件中，术语 URI(统一资源标识符)是指通过一个位置或一个名称来引用资源。当 URI 通过位置识别资源时，它被称作 URL 或统一资源定位符。当 URI 通过名称识别资源时，它就成为一个 URN 或统一资源名称。在这方面，ASP.NET MVC 旨在处理更通用的 URI，而 ASP.NET Web Forms 主要处理位置感知的物理资源。

1.1.1　模拟 ASP.NET MVC 运行时

让我们构建一个简单的 ASP.NET Web Forms 应用程序，并使用 HTTP 处理程序来理解 ASP.NET MVC 应用程序的内部机制。可以先从微软 Visual Studio 项目管理器处获取基本的 ASP.NET Web Forms 应用程序。

1. 定义可识别的 URL 的语法

由于请求的 URL 不一定与 Web 服务器上的物理文件相匹配，因此第一步即是列出哪些 URL 对应用程序有意义。为了避免过于特殊，我们假设你只支持几个固定的 URL，每一个都会映射到一个 HTTP 处理程序组件。下面的代码片段显示了需要对默认 web.config 文件所做的更改：

```
<httpHandlers>
<add verb="*"
   path="home/test/*"
```

```
    type="MvcEmule.Components.MvcEmuleHandler" />
</httpHandlers>
```

每当应用程序收到与指定 URL 匹配的请求时，它就会将其传递到指定的处理程序。

2. 定义 HTTP 处理程序的行为

在 ASP.NET 中，HTTP 处理程序是一个实现了 IHttpHandler 接口的组件。该接口比较简单，包括以下两个部分：

```
public class MvcEmuleHandler : IHttpHandler
{
   public void ProcessRequest(HttpContext context)
   {
      // Logic goes here
      ...
   }

   public Boolean IsReusable
   {
      get { return false; }
   }
}
```

大部分情况下，HTTP 处理程序都有一个只受某些通过查询字符串传递的输入数据所影响的硬编码行为。不过，我们也可以毫不费力地将处理程序用作抽象工厂来再增加一级间接层。事实上，处理程序可以使用来自请求的信息以确定要调用的外部组件以切实地处理请求。这样一来，单个 HTTP 处理程序就可以服务各种请求，并且只需要在一些专门的组件之间分配调用。

HTTP 处理程序可以解析出令牌中的 URL，并使用该信息来标识要调用的类和方法。这里是一个显示其工作原理的示例：

```
public void ProcessRequest(HttpContext context)
{
    // Parse out the URL and extract controller, action, and parameter
    var segments = context.Request.Url.Segments;
    var controller = segments[1].TrimEnd('/');
    var action = segments[2].TrimEnd('/');
    var param1 = segments[3].TrimEnd('/');

    // Complete controller class name with suffix and (default) namespace
    var fullName = String.Format("{0}.{1}Controller",
                       this.GetType().Namespace, controller);
```

```
        var controllerType = Type.GetType(fullName, true, true);

        // Get an instance of the controller
        var instance = Activator.CreateInstance(controllerType);

        // Invoke the action method on the controller instance
        var methodInfo = controllerType.GetMethod(action,
              BindingFlags.Instance |
              BindingFlags.IgnoreCase |
              BindingFlags.Public);
        var result = String.Empty;
        if (methodInfo.GetParameters().Length == 0)
        {
           result = methodInfo.Invoke(instance, null) as String;
        }
        else
        {
           result = methodInfo.Invoke(instance, new Object[] { param1 }) as String;
        }

        // Write out results
        context.Response.Write(result);
    }
```

前面的代码假定，该 URL 中服务器名称后面的第一个令牌包含一些关键信息，用来标识即将用于处理请求的专门组件。第二个令牌引用了调用该组件的方法名称。最后，第三个令牌指示要传递的参数。

3. 调用 HTTP 处理程序

给定一个 URL，比如 home/test/*，不论后面的参数是什么，home 都用于标识类，test 用于标识方法。该类的名称会进一步指定并扩展以包含一个名称空间和一个后缀。在本例中，最终的类名是 MvcEmule.Components.HomeController。该类可用于应用程序。也可公开成一个称为 Test 的方法，如下所示：

```
    namespace MvcEmule.Components
    {
        public class HomeController
        {
            public String Test(Object param1)
            {
                var message = "<html><h1>Got it! You passed '{0}'</h1></html>";
                return String.Format(message, param1);
```

```
            }
        }
    }
```

图 1-1 显示了在 ASP.NET Web Forms 应用程序中调用某个页面不可知 URL 的效果。

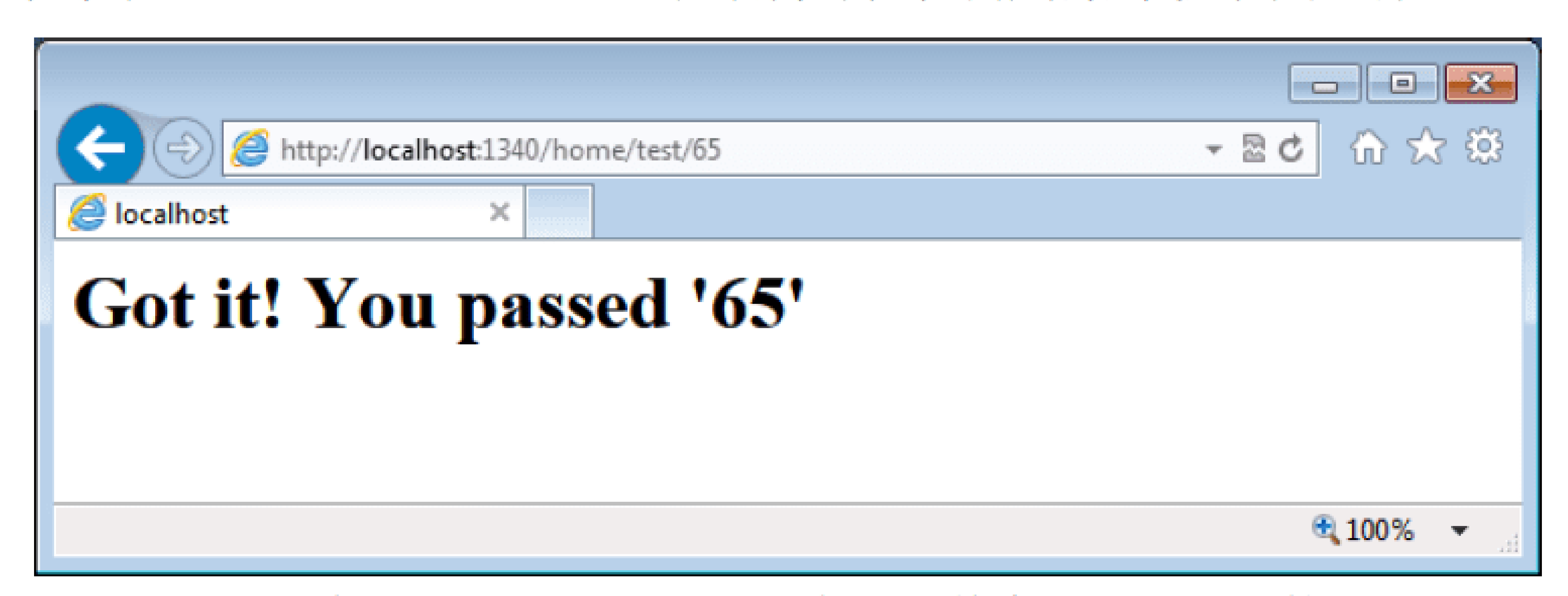


图 1-1　在 ASP.NET Web Forms 中处理某个页面不可知的 URL

这个简单的示例揭示了 ASP.NET MVC 所使用的基本机制。处理请求的专门组件是控制器。控制器是一个只有方法而无状态的类。独特的系统级 HTTP 处理程序负责将传入的请求分派到一个专门的控制器类，这样该类的实例就会执行给定的操作方法并产生响应。

那么 URL 的架构是什么呢？在本示例中，仅仅使用了硬编码的 URL。而在 ASP.NET MVC 中，可以使用非常灵活的语法来表示应用程序可识别的 URL。此外，运行时管道中的一个新系统组件会截取请求，处理该 URL，并触发 ASP.NET MVC HTTP 处理程序。该组件就是 URL 路由 HTTP 模块。

1.1.2　URL 路由 HTTP 模块

URL 路由 HTTP 模块通过查看 URL 并把它们分派到最适当的执行器来处理传入的请求。URL 路由 HTTP 模块取代了旧版本 ASP.NET 的 URL 重写功能。在其核心处，URL 重写由挂接请求、解析原始 URL 和指示 HTTP 运行时环境组成，用以处理“可能相关却不同”的 URL。

1. 取代 URL 重写

如果需要在可读的、搜索引擎优化(SEO)友好的 URL 和以编程方式处理大量的 URL 之间进行权衡，那么 URL 重写就可以发挥重要作用。比如，请思考下面的 URL：

```
http://northwind.com/news.aspx?id=1234
```

news.aspx 页面包含了检索、格式化和显示任意给定新闻所需的逻辑。检索特定新闻的 ID 是通过查询字符串上的参数提供的。作为开发人员，执行该页面再简单不过；获取查询字符串参数、运行查询并创建 HTML。而作为用户或对于搜索引擎来说，仅仅通过简单地查看该 URL 并不能真正理解该页面的意图，你不大可能轻松记住这个地址并分发出去。

URL 重写会在两个方面对你有所助益。首先，它使开发人员可使用一个通用的前端页面(如 news.aspx)来显示相关内容。第二，它使用户请求友好的 URL 成为可能，该 URL 以编程方式映射到不太直观但容易管理的 URL。总之，URL 重写确实可以将处理该请求的物理网页与所需的 URL 进行解耦。

在最新版本的 ASP.NET 4 Web Forms 中，可以使用 URL 路由将传入的 URL 与其他 URL 进行匹配而不用付出 HTTP 302 重定向的代价。相反，在 ASP.NET MVC 中，URL 路由提供了把传入的 URL 映射到控制器类和操作方法的服务。

注意：

URL 路由模块最初是作为 ASP.NET MVC 组件来开发的，而现在成为 ASP.NET 平台的一个原生部分，且如前所述，它能够同时为 ASP.NET MVC 和 ASP.NET Web Forms 应用程序提供服务，虽然是通过略微不同的 API。

2. 对请求进行路由

当一个请求在互联网信息服务(Internet Information Services，IIS)入口等待处理时，究竟会发生什么？图 1-2 为你提供了一个所涉及的各个步骤的全貌，以及 ASP.NET MVC 和 ASP.NET Web Forms 应用程序中的不同工作机制。

URL 路由模块能够为应用程序截取只能由 IIS 服务的请求。如果 URL 引用了物理文件(比如 ASPX 文件)，路由模块就会忽略该请求，除非它是以其他方式配置的。随后该请求会进入经典的 ASP.NET 机制，利用页面处理程序像寻常方式一样进行处理。

否则，URL 路由模块会尝试把请求的 URL 匹配到应用程序定义的任意路由。如果找到匹配项，请求将转到 ASP.NET MVC 空间按照控制器类的调用进行处理。如果没有找到匹配项，请求将会由标准的 ASP.NET 运行时以最佳方式来提供服务，并很可能引发一个 HTTP 404 错误。

最后，只有与预定义 URL 模式(也称为路由)相匹配的请求才能享有 ASP.NET MVC 运行时。所有这类请求会被路由到一个共同的 HTTP 处理程序，该处理程序将控制器类实例化并调用该类中一个定义了的方法。接下来，控制器方法会进而选择视图组件生成实际的响应。

3. URL 路由模块的内部结构

在执行方面，需要指出的是，URL 路由引擎是一个把 PostResolveRequestCache 事件串联起来的 HTTP 模块。在检查出请求的响应不在 ASP.NET 缓存中以后，就会触发该事件。

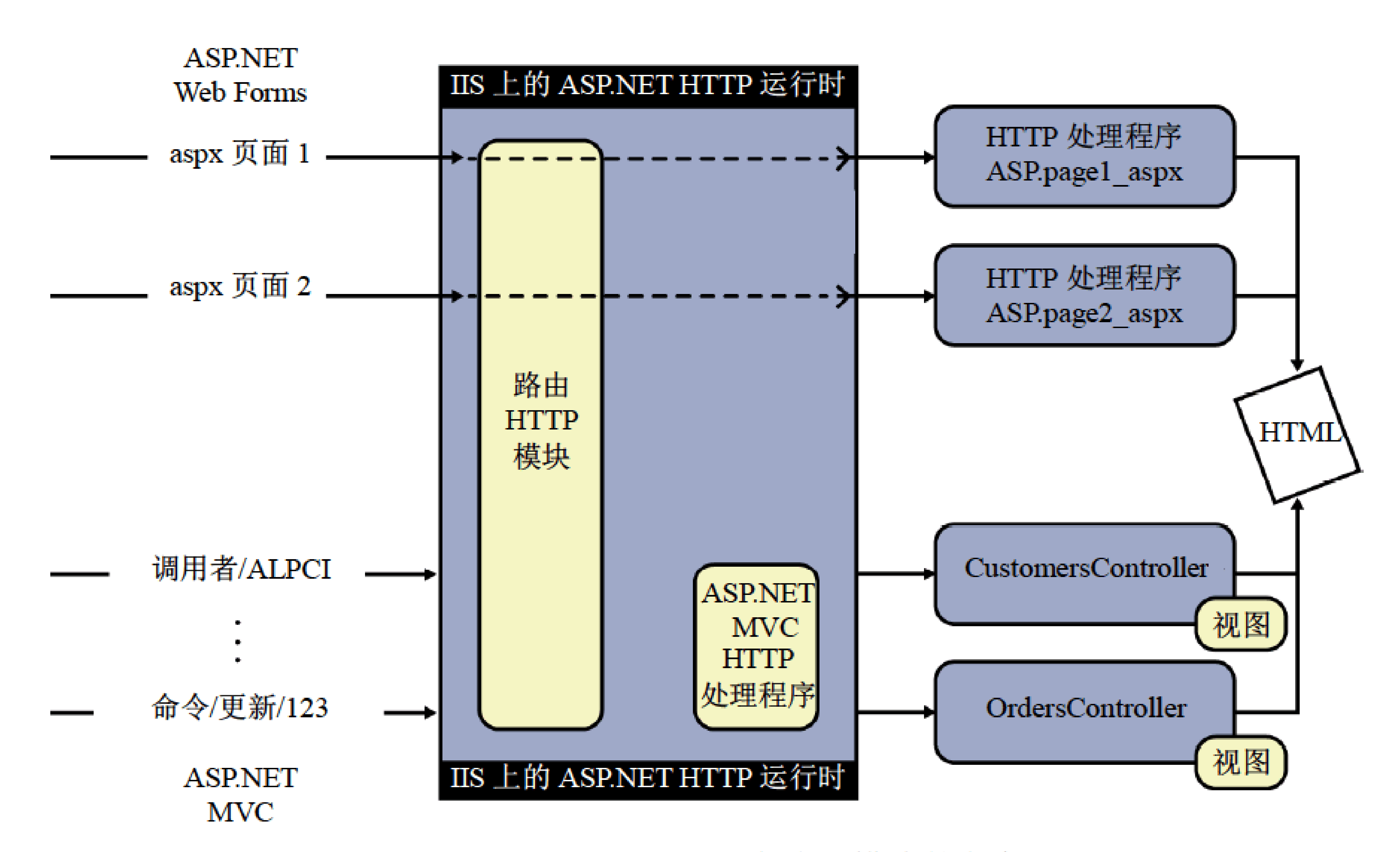


图 1-2　ASP.NET MVC 中路由模块的角色

HTTP 模块将所请求的 URL 匹配到一个用户定义的 URL 路由，并将 HTTP 上下文设置为使用 ASP.NET MVC 标准的 HTTP 处理程序来处理该请求。作为开发人员，你不可能直接处理 URL 路由模块。该模块是由系统提供的，你不需要制定任何具体的配置格式。相反，你要负责提供应用程序支持且模块会实际使用的路由。

1.1.3　应用程序路由

按照设计，ASP.NET MVC 应用程序不再强制需要依赖物理页面。在 ASP.NET MVC 中，用户要放置作用于资源的请求。然而，框架不会强行规定资源和操作的描述语法。“作用于资源”这个表述可能会让你想到具象状态传输(Representational State Transfer，REST)。当然，你这样想也并不离谱。

虽然你确实可在 ASP.NET MVC 应用程序中使用纯粹的 REST 方法，但我宁愿说 ASP.NET MVC 是松散地面向 REST 的，它确实认可诸如资源和操作的概念，但它可以让你自由使用自己的语法来表达和执行资源及操作。举例来说，在一个纯粹的 REST 解决方案中，你会利用 HTTP 动词来表示操作——GET、POST、PUT 和 DELETE——和 URL 以标识资源。在 ASP.NET MVC 中实现一个纯粹的 REST 解决方案是可能的，但你需要做一些额外工作。

ASP.NET MVC 中的默认行为是使用你自己的语法所编写的自定义 URL，通过这些语法来指定资源和操作。该语法通过 URL 模式集合来表达，也称为路由。

1. URL 模式和路由

路由是代表 URL 绝对路径的模式匹配字符串，即没有协议、服务器和端口信息的 URL 字符串。路由可以是一个常量字符串，但更有可能包含一些占位符。下面是一个示例路由：

```
/home/test
```

该路由是一个常量字符串，并只由绝对路径是/home/test 的 URL 匹配。然而大多数时候，你会处理包含一个或多个占位符的参数化路径。下面是两个示例：

```
/{resource}/{action}
/Customer/{action}
```

这两个路由都是由确切包含两个部分的任意 URL 所匹配。但后者要求第一个部分等于字符串"Customer"。而前者并不对该部分内容作具体限制。

占位符通常被称为 URL 参数，是括在大括号{}中的名称。一个路由可以有多个占位符，只要将它们用常量或分隔符分隔开来。正斜杠(/)字符用作路由各部分之间的分隔符。占位符的名称(比如 action)是代码以编程方式从实际 URL 处检索相应内容的关键所在。

下面是一个 ASP.NET MVC 应用程序的默认路由：

```
{controller}/{action}/{id}
```

该示例路由包含了由分隔符分开的三个占位符。下面的这个 URL 可匹配这个路由：

```
/Customers/Edit/ALFKI
```

可以任意添加多个路由，以及你认为合适的多个占位符。甚至还可以删除默认的路由。

2. 定义应用程序路由

应用程序路由通常在 global.asax 文件中注册，并在应用程序启动时得到处理。让我们看看 global.asax 文件中处理路由的这一部分：

```
public class MvcApplication : HttpApplication
{
   protected void Application_Start()
   {
      RouteConfig.RegisterRoutes(RouteTable.Routes);

      // Other code
      ...
   }
}
```

RegisterRoutes 是一个在通常命名为 App_:Start(但可以随意重命名该文件夹)的单独文件夹中所定义的 RouteConfig 类的方法。下面是该类的实现：

```
public class RouteConfig
{
   public static void RegisterRoutes(RouteCollection routes)
   {
      // Other code
      ...

      // Listing routes
      routes.MapRoute(
         "Default",
         "{controller}/{action}/{id}",
         new {
              controller = "Home",
              action = "Index",
              id = UrlParameter.Optional
         });
   }
}
```

如你所见，Application_Start 事件处理程序调用一个名为 RegisterRoutes 的公共静态方法，其中列出了所有路由。注意 RegisterRoutes 方法的名称及原型是随意的，可以根据实际情况更改它。

所支持的路由必须添加到由 ASP.NET MVC 管理的路由对象的静态集合中。该集合就是 RouteTable.Routes。通常使用便捷的 MapRoute 方法来填充该集合。MapRoute 方法提供了各种重载且在大部分时间都能够很好地执行。但是它不可能让你对路由对象的每个可能方面都进行配置。如果需要在路由上做一些 MapRoute 并不支持的设置，则可能需要借助于下面的代码：

```
// Create a new route and add it to the system collection
var route = new Route(...);
RouteTable.Routes.Add("NameOfTheRoute", route);
```

路由是通过一些特性来加以描述的，如名称、URL 模式、默认值、限制条件、数据令牌和路由处理程序。而最常用到的特性设置包括名称、URL 模式和默认值。让我们在你获取的默认路由的代码上进行扩展：

```
routes.MapRoute(
       "Default",
       "{controller}/{action}/{id}",
```

```
        new {
            controller = "Home",
            action = "Index",
            id = UrlParameter.Optional
        });
```

第一个参数是路由的名称；每个路由都有一个唯一的名称。第二个参数是 URL 模式。第三个参数是一个指定 URL 参数默认值的对象。

注意 URL 甚至可以匹配不完整形式的模式。让我们思考一下这个根 URL——http://yourserver.com。乍一看，此类 URL 没有匹配路由。然而，如果为 URL 参数指定一个默认值，则该部分被视为可选。因此，对于前面的示例，当你请求根 URL 时，该请求会通过调用 Home 控制器上的索引方法来解决。

3. 处理路由

ASP.NET 的 URL 路由模块在试图将传入请求的 URL 匹配到一个定义好的路由时采用了一些规则。最重要的规则是必须以在 global.asax 中所注册的顺序来检查该路由。

为确保按正确顺序处理该路由，必须将它们按照从最具体到最不具体的顺序罗列出来。在任何情况下，请记住匹配路由的搜索总是结束于首个匹配。这意味着只在列表底部添加一条新路由可能不起作用，也可能给你造成麻烦。此外，要知道在列表顶部放置一个笼统的模式将可兼容其他任何模式，不管这个模式多么具体、多么容易被遗漏。

除了列出的顺序，其他因素也会影响给 URL 匹配路由的过程。如前所述，其中一个就是你对提供给路由的默认值的设置。默认值其实就是自动分配到已定义占位符的值，以防 URL 不提供具体值。请参考下面两个路由：

```
{Orders}/{Year}/{Month}
{Orders}/{Year}
```

如果在第一条路由中，你为{Year}和{Month}都分配了默认值，那么第二条路由将永远不会参与判断，因为第一条路由有了默认值便总会得到匹配，无论 URL 是否指定年份和月份。

结尾的正斜杠(/)也是一个陷阱。路由{Orders}/{Year}和路由{Orders}/{Year}/是两个不一样的东西。相互之间不能匹配，尽管从逻辑上看起来确实不能匹配，但至少从用户的角度来看，你会期望它们是能够匹配的。

另一个影响 URL 与路由匹配的因素是，你根据需要为路由定义了约束列表。路由约束是给定 URL 参数必须满足以匹配路由的附加条件。URL 不仅应该与 URL 模式兼容，还需要包含兼容的数据。约束条件能以各种方式定义，包括通过正则表达式。下面是一个带有约束条件的路由示例：

```
routes.MapRoute(
```

```
    "ProductInfo",
    "{controller}/{productId}/{locale}",
     new { controller = "Product", action = "Index", locale="en-us" },
     new { productId = @"\d{8}",
           locale = ""[a-z]{2}-[a-z]{2}" });
```

尤其是该路由要求 productId 占位符必须是精确 8 位数的数字序列，而 locale 占位符必须是一对由短划线隔开的两个字母的字符串。约束条件并不能确保将所有的无效产品 ID 和区域设置代码都挡在门外，但至少避免了大量工作。

4. 路由处理程序

路由定义了一套最低限度的规则，根据该规则路由模块决定传入的请求 URL 是否可以被应用程序接受。最终决定如何重新映射所请求的 URL 的组件则完全是另一回事。准确地说，它是一个路由处理程序。路由处理程序是处理任意与给定路由相匹配的请求的对象。它存在的唯一目的是返回将会实际为任何匹配请求提供服务的 HTTP 处理程序。

从技术角度看，路由处理程序是一个实现了 IRouteHandler 接口的类。接口的定义如下所示：

```
public interface IRouteHandler
{
   IHttpHandler GetHttpHandler(RequestContext requestContext);
}
```

在 System.Web.Routing 名称空间中定义的 RequestContext 类，封装了请求的 HTTP 上下文以及任何可用的路由专用信息，比如 Route 对象本身、URL 参数和约束条件。这些数据被分组到一个 RouteData 对象。下面是 RequestContext 类的签名：

```
public class RequestContext
{
    public RequestContext(HttpContextBase httpContext, RouteData routeData);

    // Properties
    public HttpContextBase HttpContext { get; set; }
    public RouteData RouteData { get; set; }
}
```

ASP.NET MVC 框架并没有提供很多内置的路由处理程序，这很可能暗示了使用自定义路由处理程序的需求并不普遍。但是可扩展点是存在的，万一需要你还可以利用它。稍后将回到自定义路由处理程序的内容，并在本章后面提供一个示例。

5. 处理物理文件的请求

有助于 URL 与路由成功匹配的路由系统的另一个可配置的方面是，路由系统是否要处理匹配物理文件的请求。

默认情况下，ASP.NET 路由系统会忽略那些其 URL 可以映射到实际存在于服务器上的文件的请求。请注意，如果服务器文件存在，则路由系统将忽略该请求，即使请求匹配了路由。

如果需要，则可以通过把 RouteCollection 对象的 RouteExistingFiles 属性设置为 true 来强制路由系统处理所有的请求，如下所示：

```
// In global.asax.cs
public static void RegisterRoutes(RouteCollection routes)
{
    routes.RouteExistingFiles = true;
    ...
}
```

注意，通过路由来处理所有请求可能会在 ASP.NET MVC 应用程序中造成几个问题。比如，如果在一个示例 ASP.NET MVC 应用程序的 global.asax.cs 文件中添加上述代码并运行该应用程序，那么在访问 default.aspx 时就会立即出现一个 HTTP 404 错误。

6. 阻止已定义的 URL 路由

ASP.NET 的 URL 路由模块不会将你限制在维持可接受的 URL 列表的模式中；也可以关闭某些 URL 的路由机制。可以使用两个步骤阻止路由系统处理某些 URL。首先，为这些 URL 定义一个模式并保存到一个路由。第二，将该路由链接到一个专门的路由处理程序——StopRoutingHandler 类。其结果就是在调用 GetHttpHandler 方法时会抛出一个 NotSupported 异常。

例如，以下代码指示路由系统忽略任何.axd 请求：

```
// In global.asax.cs
public static void RegisterRoutes(RouteCollection routes)
{
  routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
  ...
}
```

IgnoreRoute 所做的就是将 StopRoutingHandler 路由处理程序关联到围绕指定 URL 模式所构建的路由上。

最后需要简单解释该 URL 中的{* pathInfo}占位符。pathInfo 令牌仅仅表示遵循.axd URL

的任意内容的占位符。但是星号(*)表示最后一个参数必须匹配该 URL 的其他部分。换句话说，遵循.axd 扩展名的所有内容都会有一个 pathInfo 参数。这种参数称为通配符参数。

7. 路由特性

包含在 ASP.NET MVC 5 中的一个受欢迎的 NuGet 包是 AttributeRouting(参见 http://attributerouting.net)。路由特性就是使用特性直接在控制器操作上定义路由。正如之前阐述过的，经典路由是基于应用程序启动时在 global.asax 中所建立的约定。

任何时候一个请求进来，URL 都会与已注册的路由模板进行匹配。如果找到匹配项，就会确定处理该请求的合适控制器和操作方法。如果没有找到，请求会被拒绝，结果通常是一个 404 消息。如今在大型应用程序中，甚至是具有较强 REST 风格的中型应用程序中，路由的数目都可能相当大，且很容易迅速达到数百的规模。你可能会很快发现，经典路由变得有点疲于处理了。出于这个原因，AttributeRouting 项目启动了，目前已经集成在 ASP.NET MVC 5 中，甚至在 Web API 中，这将在第 10 章“Web API 的执行指南”中讨论。

```
[HttpGet("orders/{orderId}/show")]
public ActionResult GetOrderById(int orderId)
{
   ...
}
```

该代码将 GetOrderById 方法设置为当 URL 模板匹配了所指定的模式时才可用于 HTTP GET 调用。路由参数——orderId 令牌——必须匹配方法签名中所定义的参数之一。有更多的一些特性可用(用于每个 HTTP 动词)，但主要的路由特性都在这里了。欲知更多信息(比如配置)，可以参阅 http://attributerouting.net，因为这个 ASP.NET MVC 的集成是现有 NuGet 包的直接体现。

1.2　控制器类

尽管在 ASP.NET MVC 名称中就明确涉及了模型-视图-控制器模式，ASP.NET MVC 的架构本质上还是集中于一个支柱——控制器。控制器控制请求的处理，并协调后端系统(例如业务层、服务、数据访问层)抓取用于响应的原始数据。接着控制器会把为请求所计算的原始数据打包成调用方的有效响应。当响应是一个标记视图时，控制器会依赖视图引擎模块将数据和视图模板结合起来，生成 HTML。

1.2.1　控制器的特征

通过 URL 路由筛选器的任何请求都被映射到一个控制器类，通过执行类上的给定方法来

进行服务。因此，控制器类是开发人员编写实际代码的地方，这些代码为请求提供所需的服务。让我们简单探讨一下控制器的一些特征。

1. 控制器的粒度

ASP.NET MVC 应用程序通常包括各式各样的控制器类。那么你应该拥有多少种控制器呢？实际数量取决于你自己，取决于你希望如何组织应用程序的操作。事实上，可以制作一个包含单个控制器类的应用程序，该控制器类用于包含任何可能请求的方法。

常见的做法包括为应用程序所实现的每个重要功能配备一个控制器类。例如，创建一个 CustomerController 类，负责与查询、删除、更新和插入客户信息等有关的请求。同样可以创建一个 ProductController 类用于处理产品信息，诸如此类等等。多数情况下，这些对象与应用程序主菜单中的项直接相关。

大体来说，控制器的粒度是一种用户界面的粒度功能。应该为每一个你在用户界面中所拥有的请求的重要来源规划一种控制器。

2. 无状态组件

所选控制器类的新实例是针对每个请求而实例化的。你可能会向类中添加的任何状态都将绑定到相同的请求生命周期中。随后控制器类必须能够从 HTTP 请求流和 HTTP 上下文中检索到它需要使用的任何数据。

3. 进一步的分层取决于你

很多时候，ASP.NET MVC 和控制器类就相当于你挥舞的一根魔杖，可以随意编写更为简洁和容易阅读与维护的分层代码。控制器类的无状态特性在这方面起了很大作用，但这还不够。

在 ASP.NET MVC 中，控制器被触发该请求的用户界面和产生浏览器视图的引擎分隔开来。控制器位于视图与系统后端之间。虽然与视图的这种分隔是很好的，并且弥补了 ASP.NET Web Forms 的薄弱点，但仅仅如此并不能确保你的代码会受到严格的关注点分离(SoC，Separation of Concerns)原则的认可。

该系统为你提供了与视图分离的最低级别——其他一切都取决于你。请记住没有什么会阻止你直接在控制器类中直接使用 ADO.NET 调用和浅显的 TRANSACT-SQL(T-SQL)语句，即使是在 ASP.NET MVC 中也不会。控制器类不是系统后端，也不是业务层。相反，它应当被视为 Web Forms 代码隐藏类的 MVC 配对物。因此，它绝对属于表示层，而非业务层。

4. 高度可测试性

控制器固有的无状态性以及与视图的简洁分离使控制器类具备了易于测试的潜力。然

而，要衡量它们的真正可测试性还应该针对其有效分层。让我们来看图 1-3。

虽然可以为控制器类提供任何你喜欢的固定输入，它的输出也可以断言没有重大问题，但操作方法的内部结构却没什么好说的。这些方法的实现与外部资源(比如数据库、服务、组件)绑定程度越紧密，控制器就越不能便捷地进行测试。

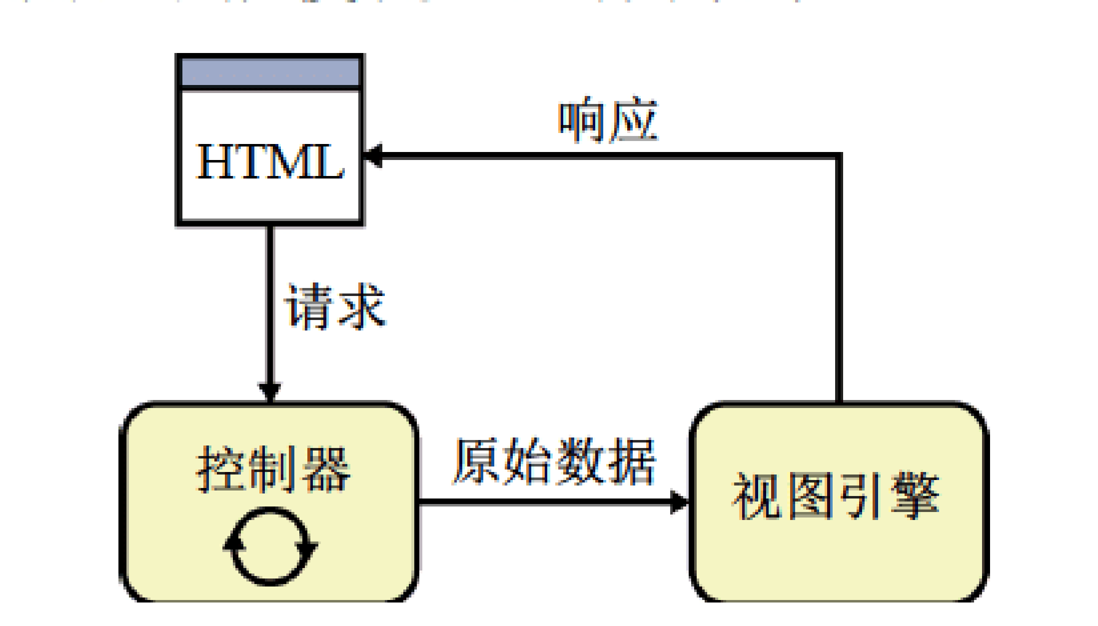


图 1-3　ASP.NET MVC 中的控制器和视图

1.2.2　编写控制器类

控制器类的编写可以概括为两个简单的步骤：第一，创建从 Controller 继承(直接或间接均可)而来的类；第二，添加一系列的公共方法。然而，必须阐明两个重要的细节：系统如何获知控制器类要实例化，以及它如何确定要调用的方法。

1. 从路由到控制器

不论你如何定义 URL 模式，任何请求都必须根据控制器名称和操作名称来解析。这是 ASP.NET MVC 的一个支柱。如果 URL 包含了一个{controller }占位符，那么控制器名称会自动从 URL 中读取。如果 URL 包含了{action}占位符，那么操作名称也将从 URL 中自动读取。

然而，缺乏这些占位符的完全自定义 URL 仍然是可以接受的。但这种情况下，需要由你通过默认值来指示控制器和操作，如下所示：

```
routes.MapRoute(
     "SampleRoute",
     "about",
     new { controller = "Home", action = "About"}
);
```

如果控制器和操作名称不能以静态方式得到解析，就可能需要编写一个自定义路由处理程序，以获得请求的详细信息，再找出控制器和操作的名称。然后只需要把它们存储在 RouteData 集合中即可，如下所示：

```
public class AboutRouteHandler : IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        if (requestContext.HttpContext.Request.Url.AbsolutePath == "/about")
```

```
            {
                requestContext.RouteData.Values["controller"] = "home";
                requestContext.RouteData.Values["action"] = "about";
            }
            return new MvcHandler(requestContext);
        }
    }
```

对于一个需要自定义处理程序的路由，其注册过程与你早先看到的有所不同。这里是你需要在 RegisterRoutes 中编写的代码：

```
public static void RegisterRoutes(RouteCollection routes)
{
    var aboutRoute = new Route("about", new AboutRouteHandler());
    routes.Add("SampleAboutRoute", aboutRoute);
    ...
}
```

请务必注意，从路由模块获得的控制器名称并不能完全匹配将被调用的类的实际名称。默认情况下，控制器类是以控制器名称加上一个 Controller 后缀来命名的。在前面的示例中，如果 home 是控制器的名称，那么该类的名称就被设定为 HomeController。注意这一约定不只适用于类的名称，也适用于名称空间。尤其是当该类会作用于默认项目名称空间下的 Controller 名称空间时。

注意：

当基于以编程方式设置控制器和操作名称的自定义路由处理程序来添加一个路由时，可能会在由 Html.ActionLink 帮助器所生成的链接方面遇到麻烦。该帮助器通常用于为用户界面的菜单和其他可视化元素创建基于路由的链接。如果用自定义处理程序添加了一个路由，你可能会惊讶地发现从帮助器获得的链接竟然是基于这个路由的。为解决这个问题，要么你把 ActionLink 更改为 RouteLink，并明确表示你希望 URL 根据哪个路由来创建，要么你在自定义路由中明确规定控制器和操作都是可选参数。

2. 从路由到操作

当 ASP.NET MVC 的运行时环境具有所选控制器类的有效实例时，它会服从操作调用程序组件以便请求的实际执行。操作调用程序会获取操作的名称并尝试将其匹配到控制器类的公共方法。

操作参数会指定要执行的操作名称。大多数情况下，控制器类已经具有一个同名的方法。如果是这样，调用程序会执行它。注意，尽管如此，你还是可以将操作名称特性关联到任何公共方法，因此需要将方法名称从操作名称解耦。下面是一个示例：

```
public class HomeController : Controller
{
    // Implicit action name: Index
    public ActionResult Index()
    {
       ...
    }

    [NonAction]
    public ActionResult About()
    {
       ...
    }

    [ActionName("About")]
    public ActionResult LoveGermanShepherds()
    {
       ...
    }
}
```

Index 方法是不用特性修饰的，所以它是以相同的名称隐式绑定到操作的。第三种公共方法有一个非常古怪的名字，但它是通过 ActionName 特性显式绑定到 About 操作的。最后，请注意如果要防止一个公共控制器方法被隐式绑定到一个操作名称，就需要使用 NonAction 特性。因此，从前面的代码段来看，当用户请求 about 操作时，就会运行 LoveGermanShepherds 方法，而不论 HTTP 动词是否曾用于设置该请求。

3. 操作和 HTTP 动词

ASP.NET MVC 足够灵活，可以让你将方法绑定到用于特定 HTTP 动词的操作。要将控制器方法与 HTTP 动词关联，可以使用参数化的 AcceptVerbs 特性，也可以使用直接特性，如 HttpGet、HttpPost 和 HttpPut 等。使用 AcceptVerbs 特性，可以指定需要哪个 HTTP 动词来执行给定的方法。让我们看看下面的示例：

```
 [AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(Customer customer)
{
   ...
}
```

对于该代码来说，其结果是 Edit 方法不能通过使用 GET 来调用。另外需要注意的是，单个方法不能有多个 AcceptVerbs 特性。如果在操作方法上添加了多个 AcceptVerbs 特性(或

类似的直接 HTTP 动词特性)，你的代码是不能编译通过的。

AcceptVerbs 特性会从 HttpVerbs 枚举类型中提取值：

```
public enum HttpVerbs
{
    Get = 1,
    Post = 2,
    Put = 4,
    Delete = 8,
    Head = 0x10
}
```

HttpVerbs 枚举由 Flags 特性修饰，所以可以通过使用按位 OR(|)运算符将来自枚举类型的多个枚举值结合到一起，同时获得另一个 HttpVerbs 值。

```
[AcceptVerbs(HttpVerbs.Post|HttpVerbs.Put)]
public ActionResult Edit(Customer customer)
{
  ...
}
```

当单击一个链接或在地址栏中键入 URL 时，就执行了一个 HTTP GET 命令。当提交一个 HTML 表单的内容时，就执行了一个 HTTP POST 命令。可以只通过 AJAX 执行其他某个 HTTP 命令，也可以从 Windows 客户端把请求发送到 ASP.NET MVC 应用程序。

将特定动词分配到给定操作方法的能力很自然地会导致重复的方法名称。具有相同名称的两种方法在控制器类中都可以被接受，只要它们接受不同的 HTTP 动词。否则，将引发异常，因为 ASP.NET MVC 不知道如何解析一词多义的情况。

注意：

还可以使用多个单独的特性，每个特性用于一个 HTTP 动词。比如 HttpGet 和 HttpPost。

4. 操作方法

让我们来看一个控制器类的示例，该示例中有一些简单而实用的操作方法：

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
       // Process input data
       ...

       // Perform expected task
```

```
        ...

        // Generate the result of the action
        return View();
    }

    public ActionResult About()
    {
        // Process input data
        ...

        // Perform expected task
        ...

        // Generate the result of the action
        return View();
    }
}
```

操作方法通过使用任意的标准 HTTP 通道攫取可用的输入数据。接着，它会准备一些操作，并可能涉及应用程序的中间层。我们可以将操作方法的模板总结如下：

- **处理输入数据**　操作方法可以从两个来源获取输入参数：由 Request 对象所公开的路由值和集合。ASP.NET MVC 并不强制要求专门的操作方法签名。然而，出于可测试性的考虑，强烈建议任何输入参数都通过签名来接收。如果可以的话，最好避免以编程方式从请求或其他来源检索输入数据的方法。你将在本章后面看到，当然在第 3 章"模型绑定架构"中会更全面地了解，存在着一个子系统——模型绑定层——用于将 HTTP 参数映射到操作方法参数。
- **执行任务**　操作方法基于输入参数进行工作，并尝试获得预期的结果。在这个过程中，操作方法可能需要与中间层交互。正如第 7 章"设计 ASP.NET MVC 控制器的注意事项"中的详尽探讨所建议的，任何交互都需要通过特设的专用服务进行。在任务结束时，任何应集成到响应中的(计算或引用的)值都会进行适当的封装。如果方法返回 JavaScript 对象标志(JavaScript Object Notation，JSON)，则数据会组成一个 JSON 序列化的对象。如果方法返回 HTML，则数据被封装成一个容器对象，并发送到视图引擎。容器对象通常也称为视图模型，可以是一对名称/值的浅显字典，或者是特定的视图强类型类。
- **生成结果**　在 ASP.NET MVC 中，控制器的方法不负责产生响应本身。不过，使用一个特别的对象(通常为视图对象)将内容提交给输出流这个过程却是由它负责的。控制器的方法会标识响应的类型(文件、普通数据、HTML、JavaScript 或 JSON)，并设置一个适当的 ActionResult 对象。

控制器的方法预期会返回一个 ActionResult 对象，或一个继承 ActionResult 类的对象。但通常情况下，控制器方法并不会直接实例化 ActionResult 对象。相反，它使用一个操作帮助器——即一个内部实例化的对象，并返回一个 ActionResult 对象。前面示例中的 View 方法提供了一个操作帮助器的最佳例子。操作帮助器方法的另一个好例子是 Json，在方法需要返回一个 JSON 字符串的时候使用。马上将会谈到这一点。

1.2.3　处理输入数据

控制器操作方法可以访问任何通过使用 HTTP 请求而提交的输入数据。输入数据可以从各种来源检索，包括表单数据、查询字符串、Cookies、路由值和提交文件。

控制器操作方法的签名是随意的。如果定义了不带参数的方法，就需要自己负责以编程方式来检索代码所需的任何输入数据。如果将参数添加到方法签名中，那么 ASP.NET MVC 会提供自动的参数解析。尤其是，ASP.NET MVC 会尝试将正式参数的名称与请求作用域字典中的命名成员相匹配，请求作用域字典会将来自查询字符串、路由、提交表单等的值连接起来。

在这一章中探讨了如何从控制器操作方法内手动检索输入数据。第 3 章将讨论自动参数解析——ASP.NET MVC 应用程序中最常见的选择。

1. 获取 Request 对象中的输入数据

在编写操作方法的主体时，你当然可以访问传给 Request 对象及其子集的任何数据，这些 Request 对象及子集包括 Form、Cookies、ServerVariables 和 QueryString。在本书的后面内容你会看到，在控制器方法的输入参数方面，ASP.NET MVC 会提供相当有吸引力的工具(如模型绑定器)，可以使你的代码更干净简洁和易于测试。话虽如此，但你依然可以编写旧样式的基于请求的代码，如下所示：

```
public ActionResult Echo()
{
   // Capture data in a manual way
   var data = Request.Params["today"] ?? String.Empty;
   ...
}
```

在 ASP.NET 中，Request.Params 字典产生于四个不同字典的组合，即 QueryString、Form、Cookies 和 ServerVariables。也可以使用 Request 对象的 Item 索引器属性，它会提供一样的功能，并按以下顺序在字典中搜索匹配项：QueryString、Form、Cookies 和 ServerVariables。下面的代码完全等同于刚刚所示的：

```
public ActionResult Echo()
{
```

```
    // Capture data in a manual way
    var data = Request["today"] ?? String.Empty;
    ...
}
```

注意对匹配项的搜索是不区分大小写的。

2. 从路由中获取输入数据

在 ASP.NET MVC 中，通常会通过 URL 提供输入的参数。这些值由路由模块捕获，并可供应用程序使用。路由值不通过 Request 对象向应用程序公开。你要使用一个稍微不同的方法以编程方式检索它们，如下所示：

```
public ActionResult Echo()
{
    // Capture data in a manual way
    var data = RouteData.Values["data"] ?? String.Empty;
    ...
}
```

路由数据是通过 Controller 类的 RouteData 属性公开的。同样，在这种情况下，对匹配项的搜索不用区分大小写。

RouteData.Values 字典是一个字符串/对象字典。大多数时候该字典只包含字符串。但是，如果以编程方式填充该字典(比如通过自定义路由处理程序)，那么它就可以包含其他类型的值。这种情况下，需要手动进行必要的类型强制转换。

3. 从多个来源获取输入数据

当然，可以在同一控制器方法中混合使用 RouteData 和 Request 调用。举例来说，让我们考虑下面的路由：

```
routes.MapRoute(
      "EchoRoute",
      "echo/{data}",
      new { controller = "Home", action = "Echo", data = UrlParameter.Optional }
);
```

http://yourserver/echo/Sunday 是一个有效的 URL。接下来显示的代码将很容易攫取到数据参数(Sunday)的值。以下是 HomeController 类的 Echo 方法的一种可能实现：

```
public ActionResult Echo()
{
    // Capture data in a manual way
    var data = RouteData.Values["data"];
```

```
    ...
}
```

如果调用下面的 URL，又会怎样？

```
http://yourserver/echo?today=3/27/2011
```

该 URL 仍然会匹配路由模式，但它不提供数据参数的值。不过，出于控制器操作的考虑，URL 会在查询字符串中添加一些输入值。下面是支持这两种方案的修订版 Echo 方法：

```
public ActionResult Echo()
{
   // Capture data in a manual way
   var data = RouteData.Values["data"] ??
              (Request.Params["today"] ?? String.Empty);
   ...
}
```

问题是，"我应该为每个可能的输入通道都规划一个不同的代码分支吗，比如表单数据、查询字符串、路由、Cookies 等？"让我们继续阅读 ValueProvider 字典一节。

4. ValueProvider 字典

在 Controller 类中，ValueProvider 属性只会为从各种来源收集到的输入数据提供单个容器。默认情况下，ValueProvider 字典由来自下列源的输入值(按特定顺序)填充：

(1) **子操作值**　输入值由子操作方法调用来提供。子操作是对来源于视图的控制器方法的调用。当视图回调控制器以获取额外数据或要求执行一些可能会影响正在提交的输出的特殊任务时，就会发生子操作调用。相关内容将会在第 2 章"ASP.NET MVC 视图"中探讨。

(2) **表单数据**　输入值由提交的 HTML 表单中所输入字段的内容提供。该内容与你通过 Request.Form 获得的内容相同。

(3) **路由数据**　输入值由与当前所选路由中所定义的参数相关联的内容提供。

(4) **查询字符串**　输入值由当前 URL 的查询字符串中所指定的参数内容提供。

(5) **提交的文件**　输入值在当前请求的上下文中通过 HTTP 提交的文件来表示。

ValueProvider 字典提供了一个以 GetValue 方法为中心的自定义编程接口。下面是一个示例：

```
var result = ValueProvider.GetValue("data");
```

要知道，GetValue 不会返回 String 或 Object 类型。相反，它会返回 ValueProviderResult 类型的一个实例。该类型有两个用于实际读取真实参数值的属性：RawValue 和 AttemptedValue。前者是 Object 类型，包含由来源所提供的原始值。AttemptedValue 属性却是一个字符串，代

表了强制转换为 String 类型的结果。下面是如何使用 ValueProvider 来实现 Echo 方法的示例代码。

```
public ActionResult Echo()
{
   var data = ValueProvider.GetValue("data").AttemptedValue ??
              (ValueProvider.GetValue("today").AttemptedValue ??
              String.Empty);
   ...
}
```

说到参数名称，ValueProvider 比 Request 和 RouteData 的要求更高一些。如果参数的大小写输入错误，会得到一个从 GetValue 返回的 null 对象。如果之后只是读取值而不检查为空的结果对象，就会导致一个异常。

最后要注意的是，默认情况下你不能通过 ValueProvider 字典获得对 Cookies 的访问权。然而，通过定义一个实现 IValueProvider 接口的类，值提供器列表可以以编程方式得到扩展。

注意：

值提供器机制可用于检索一些封装到合适的值集合的请求数据。默认的值提供器将你从查询 QueryString 或 Form 集合的负担中解救了出来。如果需要从一个 Cookie 或一个请求标头中读取数据呢？可以走传统路子，读取 Request 对象的 Header 或 Cookies 集合，并编写代码提取单独的值。然而，如果应用程序广泛地基于请求标头或 Cookies，那么就可能需要考虑编写一个自定义的值提供器了。从社区网站上不难发现这两种情况的示例。可以在 http://blog.donnfelker.com/2011/02/16/asp-net-mvc-building-web-apis-with-headervalueprovider 找到公开请求标头的值提供器的一个好例子。

1.2.4　产生操作结果

一个操作方法可以产生多种结果。例如，一个操作方法可以只用作 Web 服务，并在对请求的响应中返回普通字符串或 JSON 字符串。同样，操作方法可以确定是否不返回内容或需要重定向到另一个 URL。这两种情况下，浏览器都只会得到一个 HTTP 响应，而没有太多内容。这就是说产生操作的原始结果(例如从中间层收集值)是一方面；对原始结果进行处理生成用于浏览器的实质 HTTP 响应是另一方面。ActionResult 类代表了用于实现该编程方面的 ASP.NET MVC 基础架构。

1. ActionResult 类的内部结构

操作方法通常会返回一个 ActionResult 类型对象。但 ActionResult 类型并不是一个数据容器。更确切地说，它是一个抽象类，提供通用编程界面，代表操作方法执行进一步的操作。

下面是 ActionResult 类的定义：

```
public abstract class ActionResult
{
    protected ActionResult()
    {
    }

    public abstract void ExecuteResult(ControllerContext context);
}
```

通过重写 ExecuteResult 方法，派生类会获得访问任何由操作方法的执行所产生的数据的权限，并触发一些后续操作。一般来说，这些后续操作与一些浏览器的响应生成有关。

2. 预定义操作结果类型

由于 ActionResult 是一个抽象类型，因此每个操作方法实际上都要求返回一个更具体类型的实例。表 1-1 列出了所有预定义的操作结果类型。

表 1-1　ASP.NET MVC 中的预定义 ActionResult 类型

类　型	描　述
ContentResult	将原始内容(不一定是 HTML)发送给浏览器。该类的 ExecuteResult 方法会将接收的所有内容序列化
EmptyResult	不向浏览器发送内容。该类的 ExecuteResult 方法不处理任何任务
FileContentResult	将一个文件的内容发送给浏览器。该文件的内容会表示成一个字节数组。ExecuteResult 方法仅会将该字节数组写入输出流中
FilePathResult	将一个文件的内容发送给浏览器。该文件由其路径和内容类型进行标识。ExecuteResult 方法会在 HttpResponse 时调用 TransmitFile 方法
FileStreamResult	将一个文件的内容发送给浏览器。该文件的内容会借助一个 Stream 对象来表示。ExecuteResult 方法会将提供的文件流复制到输出流
HttpNotFoundResult	将一个 HTTP 404 响应代码发送给浏览器。该 HTTP 状态码标识一个请求失败了，原因是该请求的资源未找到
HttpUnauthorizedResult	将一个 HTTP 401 响应代码发送给浏览器。该 HTTP 状态码标识一个未授权的请求
JavaScriptResult	将 JavaScript 文本发送给浏览器。该类的 ExecuteResult 方法会输出该脚本并设置相应的内容类型
JsonResult	将一个 JSON 字符串发送给浏览器。该类的 ExecuteResult 方法会为应用程序或 JSON 设置内容类型，并调用 JavaScriptSerializer 类为 JSON 序列化提供的托管对象

(续表)

类　　型	描　　述
PartialViewResult	将 HTML 内容发送给浏览器，它代表整个页面视图的一个片段。ASP.NET MVC 中视图片段的概念与 Web Forms 中用户控件的概念非常相似
RedirectResult	将一个 HTTP 302 响应代码发送给浏览器以将浏览器导航到指定的 URL。该类的 ExecuteResult 方法只调用 Response.Redirect
RedirectToRouteResult	就像 RedirectResult，RedirectToRouteResult 会将一个 HTTP 302 响应代码发送给浏览器并重定向到一个新的 URL。不同之处在于用来确定目标 URL 所采用的逻辑和输入数据。这种情况下，URL 是基于操作/控制器配对或路由名称来构建的
ViewResult	将代表完整页面视图的 HTML 内容发送给浏览器

注意，FileContentResult、FilePathResult 和 FileStreamResult 是从同一基类派生而来的，该基类是：FileResult。如果要以下载的某些文件内容或者一些表示为字节数组的普通二进制内容来回复请求，那么可以使用任何一个上述操作结果对象。PartialViewResult 和 ViewResult 是从 ViewResultBase 继承而来的，并会返回 HTML 内容。最后，HttpUnauthorizedResult 和 HttpNotFoundResult 分别代表对未经授权的访问与缺少资源这两种常见情况的响应。这两者都从可进一步扩展的 HttpStatusCodeResult 类派生而来。

3. 执行操作结果的机制

为了更好地理解操作结果类的机制，让我们剖析一个预定义类。比如 JavaScriptResult 类，它提供了一些简单而有意义的行为。JavaScriptResult 类表示返回一些脚本到浏览器的操作。下面是一个提供了 JavaScript 代码的可能操作方法：

```
public JavaScriptResult GetScript()
{
    var script = "alert('Hello')";
    return JavaScript(script);
}
```

在这个示例中，JavaScript 是 Controller 类的一个帮助器方法，充当了 JavaScriptResult 对象工厂的角色。其实现如下：

```
protected JavaScriptResult JavaScript(string script)
{
   return new JavaScriptResult() { Script = script };
}
```

JavaScriptResult 类提供了一个公共属性——Script 属性——它包含了会写入输出流的脚本

代码。其实现如下：

```
public class JavaScriptResult : ActionResult
{
   public String Script { get; set; }

   public override void ExecuteResult(ControllerContext context)
   {
      if (context == null)
         throw new ArgumentNullException("context");

      // Prepare the response
      HttpResponseBase response = context.HttpContext.Response;
      response.ContentType = "application/x-javascript";
      if (Script != null)
         response.Write(Script);
   }
}
```

正如你所看到的，ActionResult 类的最终目的是准备好要返回到浏览器的 HttpResponse 对象。这就需要设置内容类型、过期策略、标头以及内容。

4. 返回 HTML 标记

大部分时候，请求是通过发回 HTML 标记来处理的。构成用于浏览器的 HTML 是 Web 框架的核心所在。在 ASP.NET Web Forms 中，构成 HTML 的任务是通过页面完成的。开发人员创建 ASPX 页面作为视图模板和代码隐藏类的混合容器。抓取结果的操作与实际响应的产生，这两者在单一的运行时环境中是难以分开的。在 ASP.NET MVC 中，产生出结果是操作方法的责任；管理响应的构成和服务则是框架的责任。而最后，构成 HTML 标记更是另一个系统组件——视图引擎——的责任。

第 2 章将介绍视图引擎，现在只能说视图引擎知道如何为给定的操作检索视图模板，也知道如何将其处理成混合了模板信息和原始数据的普通 HTML 流。视图引擎规定了视图模板的语法(比如 ASPX、Razor 和 Spark)；而由开发人员决定合并到视图的原始数据格式。让我们看一个返回 HTML 的示例操作方法：

```
public ActionResult Index()
{
  return View(); // same as View("index");
}
```

View 方法是一个帮助器方法，负责创建 ViewResult 对象。ViewResult 对象需要了解视图模板、可选的母版视图以及将集成到最终 HTML 的原始数据。在上述代码段中，View 没有

参数并不意味着没有数据在实际传递。下面是该方法签名中的一个：

```
protected ViewResult View(String viewName, String masterName, Object model)
```

按照约定，视图模板是一个以操作名称(本例中是 Index)命名的文件，并位于特定的文件夹中。其确切位置取决于当前所选视图引擎的实现。默认情况下，视图模板应该位于与控制器名称相匹配的目录之下的 Views 文件夹中——比如说，Views/Home。请注意当你部署网站时必须保持这种目录结构。

视图模板文件的扩展名也取决于视图引擎的实现。对于 ASP.NET MVC 中的两个预定义的视图引擎来说，如果选择 ASPX 视图引擎，那么扩展名为.aspx；如果选择 Razor 视图引擎，则扩展名为.cshtml(或.vbhtml)(将在第 2 章提供更多的相关内容)。

5. 返回 JSON 内容

ASP.NET MVC 非常适合实现那些会从 Ajax 上下文的 jQuery 代码片段中回调的简单 Web 服务。你只需要设置一个或多个操作方法，以返回 JSON 字符串而不是 HTML。下面有一个示例：

```
public JsonResult GetCustomers()
{
    // Grab some data to return
    var customers = _customerRepository.GetAll();

    // Serialize to JSON and return
    return Json(customers);
}
```

Json 帮助器方法可以获取一个普通的.NET 对象，并使用内置的 JavaScriptSerializer 类将它序列化为字符串。

注意：

如果控制器操作方法不返回 ActionResult 呢？首先且最重要的前提是，没有抛出异常。简单地说，ASP.NET MVC 会将任何从操作方法返回的值(数字、字符串或自定义对象)封装成一个 ContentResult 对象。ContentResult 对象的执行会导致浏览器中值的序列化。例如，一个返回整数或字符串的操作会使你得到一个照原样显示数据的浏览器页面。返回自定义对象却会显示由该对象的 ToString 方法的执行所产生的任何字符串。如果该方法返回 HTML 字符串，那么没有任何标记会自动编码，浏览器也不可能正确地解析它。最后，空白返回值实际上是被映射到了 EmptyResult 对象，而它的执行不会触发任何操作。

控制器内的异步操作

控制器的主要目的是服务于用户界面的需要。需要执行的任何服务器端函数都要映射到一个控制器方法并从用户界面触发。执行完自己的任务后，控制器方法会选择下一个视图、封装一些数据并指示数据的提交。

这是控制器行为的本质。然而，控制器中的其他特征也往往是必要的，尤其是当控制器应用于大而复杂且带有特别需求的新型系统时，比如长时间运行的需求。在 ASP.NET MVC 的早期版本中，你需要遵循特定的模式，来赋予控制器方法的异步行为。从.NET Framework 4.5 开始，就可以得益于新的 async/await 语言结构和底层的.NET 机制了。下面是用一个或多个异步方法来编写一个控制器类的示例：

```
public class HomeController : AsyncController
{
    public async Task<ActionResult> Rss()
    {
        // Run the potentially lengthy operation
        var client = new HttpClient();
        var rss = await client.GetStringAsync(someRssUrl);

        // Parse RSS and build the view model
        var model = new HomeIndexModel();
        model.News = ParseRssInternal(rss);
        return model;
    }
}
```

该代码像是编写来用以同步运行的——你不必关心回调问题。不过最终，它非常具有可读性并且以异步方式运行，这都得益于在你使用 async/await 关键字时，C#编译器添加的语法糖分。

1.3　本章小结

控制器是 ASP.NET MVC 应用程序的核心。它在用户请求与服务器系统功能之间扮演媒介的角色。控制器与用户界面操作连接，并与中间层保持联系。控制器会安排页面的呈现，但其自身并不运行任何呈现方面的任务。这是与 ASP.NET Web Forms 的一个主要区别。在控制器中，请求的处理与显示是完全分开的。而在 Web Forms 中，页面处理阶段却集成了某些任务的执行与响应的呈现。

虽然是基于不同的语法，但控制器方法与 ASP.NET Web Forms 中的回传事件处理程序并

没有太多不同之处。在这方面，控制器类扮演着与 Web Forms 中代码隐藏类相同的角色。控制器以及 Web Forms 代码隐藏类都归属于表现层。为此，你势必要关注如何对各种不同操作方法的行为进行编码。请记住，在 ASP.NET MVC 中，解决方案建筑中的任何分层都取决于你。

本章跳过了在控制器方法中添加行为的所有细节。重点关注在这之前部分和之后部分的内容概述。在第 2 章中，将深入到其后的部分；因此重点将会是视图、视图引擎和标记的生成。然后在第 3 章中，会探讨模型绑定以及在操作方法的特性发挥作用之前所发生的事情。在第 7 章，将重新回到本章的主题，针对如何在控制器类中构造方法这一问题，做一些设计上的考量。

我们刚刚只是通过了控制器的第一道关口。还有很多需要讲解和学习的内容。

第 2 章

ASP.NET MVC 视图

设计并非外观怎样，感觉如何。设计是(解决)如何工作的问题。

——*Steve Jobs*

在 ASP.NET MVC 中，任何请求都是通过一些控制器上正在执行的操作来处理的。这一点甚至对于初学者来说也是比较容易理解的。但是，请求又有着初学者通常难以把握的另一面——生成用于浏览器的 HTML。

在 ASP.NET Web Forms 中，你甚至不会去考虑操作的事情——你考虑的是一个页面，而一个页面会同时包含逻辑和视图。在经典的 ASP.NET 中，我们假设首先要制作一个 register.aspx 页面，用户可以点击一个链接访问到该页面。该页面会展开它的用户界面，其界面结尾处是一个提交按钮。该按钮会引发一个 POST 并提交给负责处理发送数据的页面，将应用程序调整到适当的状态，并准备预期的感谢页面。整个过程都源于该页面的资源。

然而，在 ASP.NET MVC 中，你要将注册操作设置在一些控制器类上。当通过 GET 命令调用该操作时，用于数据输入的用户界面就会显示出来。在通过 POST 进行调用时，该操作会执行所需的服务器端任务，然后设法处理并返回感谢界面。整个工作流程类似于在非网络环境中处理的过程。

在 ASP.NET MVC 中，你只需要处理两种主要类型的组件。一种是控制器，它负责执行请求并为原始输入生成原始结果。另一种是视图引擎，它负责生成基于由控制器计算出的结果的任何预期的 HTML 响应。在这一章中，首先我会简要地论述视图引擎的内部结构，然后对如何为引擎提供视图模板和数据做一些实际的考量。

注意：

正如在第 1 章“ASP.NET MVC 控制器”中所揭示的那样，控制器操作不一定非得生成一些 HTML。可以把 ASP.NET MVC 应用程序看作是能够服务于各种响应的组件集合，这些响应包括 HTML、JavaScript、JavaScript 对象标志(JSON)和纯文本。在这一章中，我会将讲解内容限制在负责生成 HTML 的子系统上。本书的后续内容会详述其他类型的响应。

2.1　视图引擎的结构与性能

视图引擎是为浏览器实际生成 HTML 输出的组件。视图引擎负责为每个请求返回 HTML，并且它通过将视图模板和由控制器传递进来的数据进行融合来准备其输出。该模板以一种引擎专用的标记语言来表示；其数据在字典或强类型对象中进行封装传递。图 2-1 显示了视图引擎与控制器协同工作的整体状况。

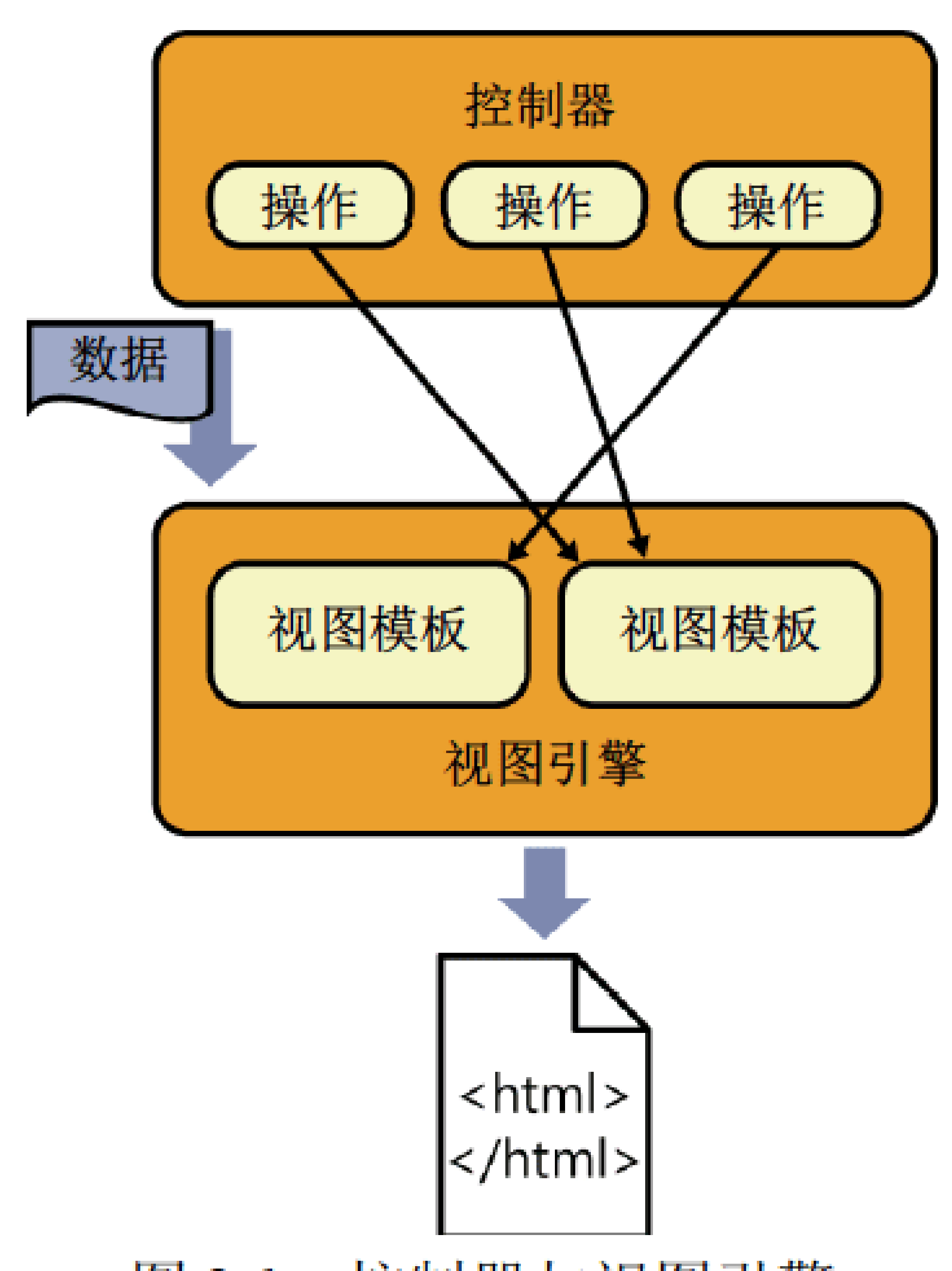


图 2-1　控制器与视图引擎

2.1.1　视图引擎的机制

在 ASP.NET MVC 中，视图引擎只是一个实现固定接口——IViewEngine 接口的类。每个应用程序可以有一个或多个视图引擎。在 ASP.NET MVC 5 中，每个应用程序默认配备两个视图引擎。接下来让我们进行更详细的了解。

1. 检测已注册的视图引擎

直到 ASP.NET MVC 4，首次创建 ASP.NET MVC 应用程序的时候，微软 Visual Studio 项目向导才允许你选择自己喜欢的视图引擎——ASPX 或 Razor。图 2-2 显示了在 Visual Studio 2013 中安装 ASP.NET MVC 5 时出现的特定对话框。

如你所见，没有 ASPX 和 Razor 之间的选项，且所有视图文件都会使用 Razor 标记语言来自动创建。除了外观，你在此处所做的选择对应用程序的影响有限。实际上，你的选择只会影响向导为你所创建的项目文件内容。默认情况下，任何 ASP.NET MVC 5 应用程序都总是会加载两个视图引擎：Razor 和 ASPX。

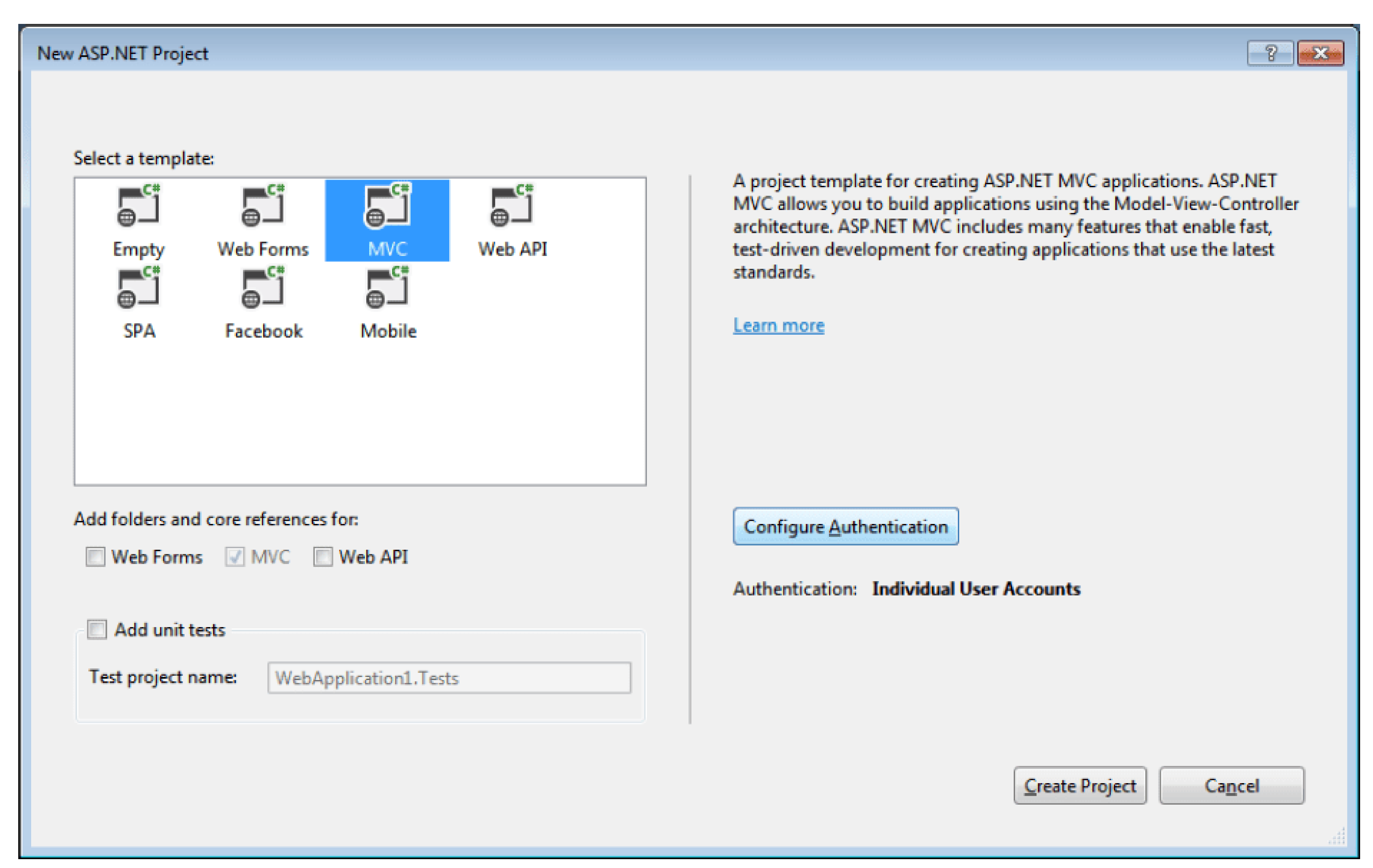


图 2-2　选择你喜欢的视图引擎

ViewEngines 类是跟踪当前已安装引擎的系统资源库。这个类比较简单，只公开了一个名为 Engine 的静态集合成员。该静态成员是由这两个默认引擎初始化的。下面是一个出自该类的代码摘录：

```
public static class ViewEngines
{
   private static readonly ViewEngineCollection _engines =
          new ViewEngineCollection {
                new WebFormViewEngine(),
                new RazorViewEngine()
          };

   public static ViewEngineCollection Engines
   {
      get { return _engines; }
   }
   ...
}
```

假如你对使用 ViewEngines.Engines 以编程方式检测已安装引擎感兴趣，下面显示了其工作原理：

```
private static IList<String> GetRegisteredViewEngines()
{
```

```
    return ViewEngines
            .Engines
            .Select(engine => engine.ToString())
            .ToList();
}
```

最可能面临使用 ViewEngines.Engines 的情况是当你需要添加一个新的视图引擎或卸载现有视图引擎的时候。这需要在应用程序启动时进行，更准确地说，是在 global.asax 里的 Application_Start 事件中。

2. 解构视图引擎

视图引擎是一个实现 IViewEngine 接口的类。该接口的协议表明其与引擎预期会提供的所有服务有关：引擎会以 ASP.NET MVC 基础架构的名义负责检索(部分)视图对象。视图对象代表了在 ASP.NET MVC 中构建实际 HTML 响应所需的所有信息的容器。下面是其接口成员：

```
public interface IViewEngine
{
    ViewEngineResult FindPartialView(
         ControllerContext controllerContext,
         String partialViewName,
         Boolean useCache);
    ViewEngineResult FindView(
         ControllerContext controllerContext,
         String viewName,
         String masterName,
         Boolean useCache);
    void ReleaseView(
         ControllerContext controllerContext,
         IView view);
}
```

表 2-1 描述了 IViewEngine 接口中方法的作用。

表 2-1　IViewEngine 接口的方法

方　法	描　述
FindPartialView	创建并返回一个表示一段 HTML 代码的视图对象
FindView	创建并返回一个表示一个 HTML 页面的视图对象
ReleaseView	释放指定的视图对象

FindPartialView 和 FindView 都会返回一个 ViewEngineResult 对象，它表示了在服务器目录树周围找出一个用于视图的模板并将该模板实例化的结果。以下内容是该类的签名：

```
public class ViewEngineResult
{
    ...

    // Members
    public IEnumerable<String> SearchedLocations { get; private set; }
    public IView View { get; private set; }
    public IViewEngine ViewEngine { get; private set; }
}
```

ViewEngineResult 类型只聚合了三个元素：视图对象、用于创建视图对象的视图引擎对象，以及搜索到的用来查找视图模板的位置列表。SearchedLocations 属性的内容取决于所选视图引擎的结构和性能。ReleaseView 方法旨在释放视图对象已经使用过的所有引用。

3. 视图引擎的调用方是谁

虽然图 2-1 似乎显示出控制器与视图引擎之间有直接联系，但这两个组成部分从不直接通信。相反，控制器和视图引擎的活动都由一个外部的管理器对象(操作调用程序)进行协调。操作调用程序由负责处理请求的 HTTP 处理程序直接触发。操作调用程序主要做两件事。第一，执行控制器的方法并保存操作结果。第二，处理操作结果。图 2-3 给出了一个序列图。

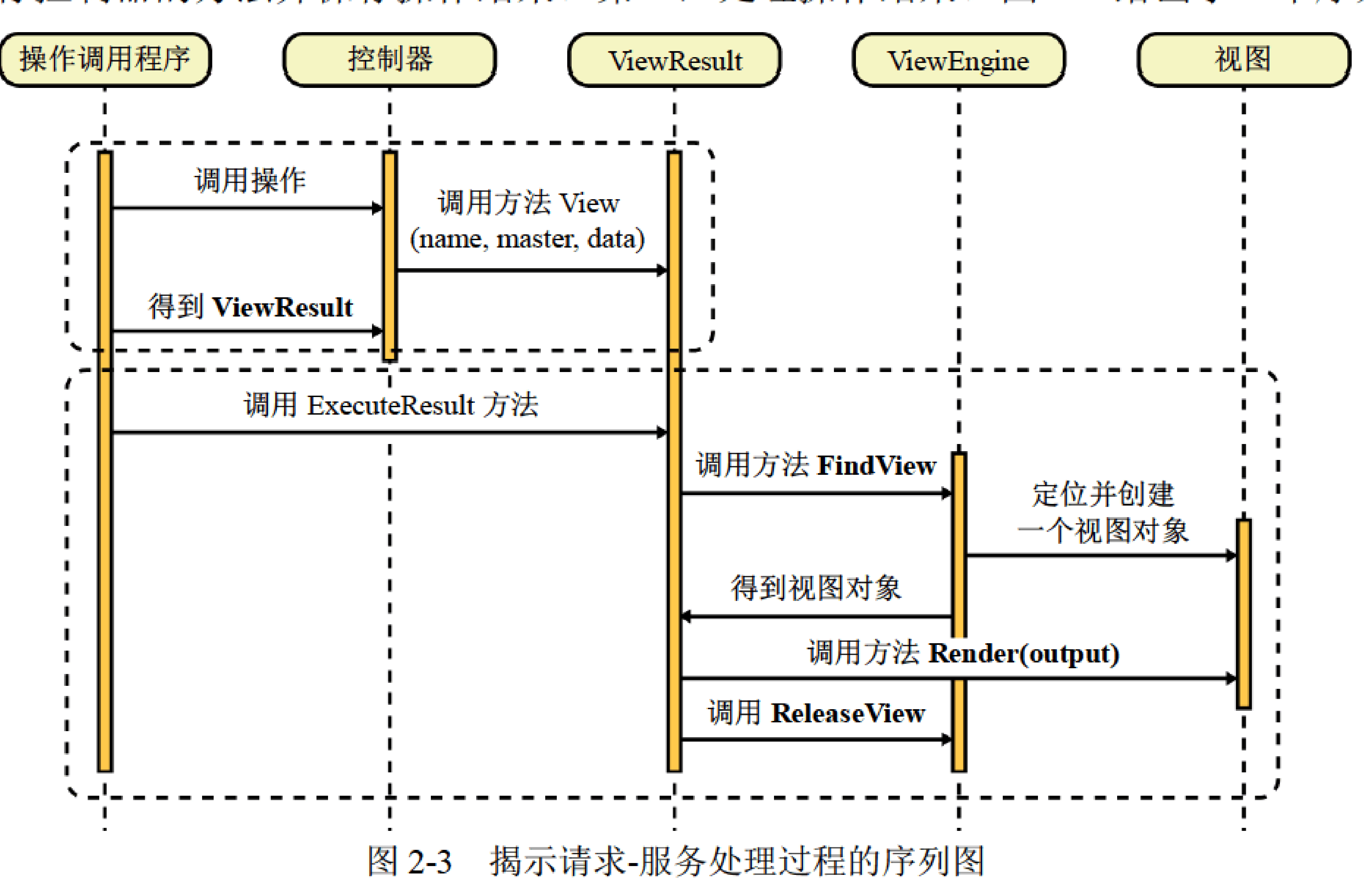


图 2-3　揭示请求-服务处理过程的序列图

让我们思考一个控制器方法的典型代码，正如你在第 1 章中所见的。

```
public ActionResult Index()
{
    // Some significant code here
    ...

    // Order rendering of the next view
    return View();
}
```

控制器类上的 View 方法会将几段数据一起封装进一个单独的容器：ViewResult 类中。其信息包括视图模板名称，该名称对应的视图就是控制器已经选择好的下一个要向用户显示的视图。母版视图名称的数据可以选择放进或不放进 ViewResult 中。最后，ViewResult 容器还会合并将在视图中显示的计算得出的数据。当 View 方法未获得任何参数时，就像在之前所显示的代码片段中那样的，则将提供默认值。ViewResult 对象的实例会被传回给操作调用程序。

接下来，操作调用程序会在 ViewResult 对象上调用 ExecuteResult 方法。该方法会通过已注册视图引擎的列表，找到一个可以匹配指定视图和母版视图名称的视图引擎。如果没有找到，则会抛出一个异常。否则，选定的视图引擎会被要求创建一个基于所提供信息的视图对象。

随后，ViewResult 对象会命令视图将内容呈现给所提供的流——实际的响应流。最后，ViewResult 对象将指示视图引擎释放该视图。

4. 视图对象

视图对象是实现 IView 接口的类的实例。视图对象的唯一目的是编写一些 HTML 响应到文本编写器。每个视图都由名称来标识。视图的名称同时也与某些定义了用于呈现 HTML 布局的物理文件相关联。视图名称与实际 HTML 布局两者之间的关联由视图引擎负责处理。

视图的名称是控制器操作上 View 方法应该提供的参数之一。如果程序员没有显式定义这样的参数，则系统会按照惯例假定视图名称与操作名称相同(第 1 章中介绍过，操作名称未必与方法名称相匹配)。

IView 接口如下所示：

```
public interface IView
{
    void Render(ViewContext viewContext, TextWriter writer);
}
```

从内部来看，视图对象是一个封装器，它封装了对没有数据的可视化布局进行描述的对

象。视图呈现意味着以数据填充布局并将其以 HTML 的形式呈现给某些流。在 ASP.NET MVC 中，两个默认视图引擎中的 ASPX 视图引擎——仅使用 ASP.NET Page 派生类来表示可视化布局。另一个视图引擎——Razor 引擎——则是建立在围绕同一核心理念设计的不同类的基础上的。Razor 引擎使用了与 ASP.NET Page 类对应的 WebMatrix①。

2.1.2　视图模板定义

在 ASP.NET MVC 中，显示给用户的一切内容都由视图所产生，并且依据模板文件进行描述。之后图形布局会转化为 HTML，并通过一个或多个级联样式表(CSS)文件设置样式。但是模板文件的写法却取决于视图引擎。每个视图引擎都有其自有的定义模板的标记语言，也有将视图名称解析到模板文件的规则集。

1. 模板解析

在任务处理结束时，控制器要找出呈现给用户的下一个视图的名称。但是视图名称必须转化为一些合适的 HTML 标记。这需要额外的几个步骤。首先，系统需要确定哪个视图引擎(如果有的话)可以成功处理该视图的请求。其次，视图名称必须匹配到一个 HTML 布局，并基于该布局成功地创建一个视图对象。

从获知视图名称时开始，ViewResult 对象(在前面的图 2-3 中所示)就会按照视图引擎出现在 ViewEngines.Engines 集合中的顺序，对所有已安装的视图引擎进行查询。每个视图引擎都会被检索以确定其是否有能力呈现一个给定名称的视图。

按照惯例，每个 ASP.NET MVC 视图引擎都使用它们自己的算法将视图名称转化成引用最终 HTML 标记的资源名称。对于两个预定义的视图引擎，搜索算法会尝试将视图名称与位于固定磁盘路径中的一个物理文件进行匹配。

不过，自定义视图引擎可以摆脱这些限制并采用一组不同的规则。例如，它可以从数据库中加载视图布局，或者它可以使用一组自定义的文件夹。

2. 默认规则和文件夹

ASPX 和 Razor 视图引擎使用相同的核心规则来解析视图名称。两者都会将视图名称与文件名称相匹配，并且两者都期望在同一组预定义文件夹中找到这些文件。ASPX 和 Razor 的唯一区别在于包含视图布局的文件的扩展名。

① 微软 WebMatrix 是一个全新的 Web 开发平台。区别于现有的开发平台，WebMatrix 的特点是一站式和简化的开发过程，您只需要花几个小时便可学会使用 WebMatrix、CSS、HTML、HTML5、ASP.NET、SQL、数据库等知识以及如何编写简单的 Web 应用程序。这个工具可以免费使用，提供了核心代码和数据库支持，集成了一个开元 Web 应用程序库，以及可以直接发布/部署站点的强大工具。与庞大的 Visual Studio 或 Visual Web Developer 相比，WebMatrix 体积只有 15MB 而已，开发人员可以很快速地开始 ASP.NET、PHP 站点的开发和发布。

除非你安装了自定义视图引擎，否则 ASP.NET MVC 应用程序会在 View 文件夹中查找其视图模板。从图 2-4 中可以看出，View 文件夹包含很多子文件夹，每个都以一个现有的控制器名称命名。最后，控制器文件夹包含了其名称预期会匹配视图名称的物理文件，同时它的扩展名必须是可用于 ASPX 视图引擎的.aspx 和可用于 Razor 视图引擎的.cshtml(如果在微软 Visual Basic 中编写 ASP.NET MVC 应用程序，则扩展名是.vbhtml)。

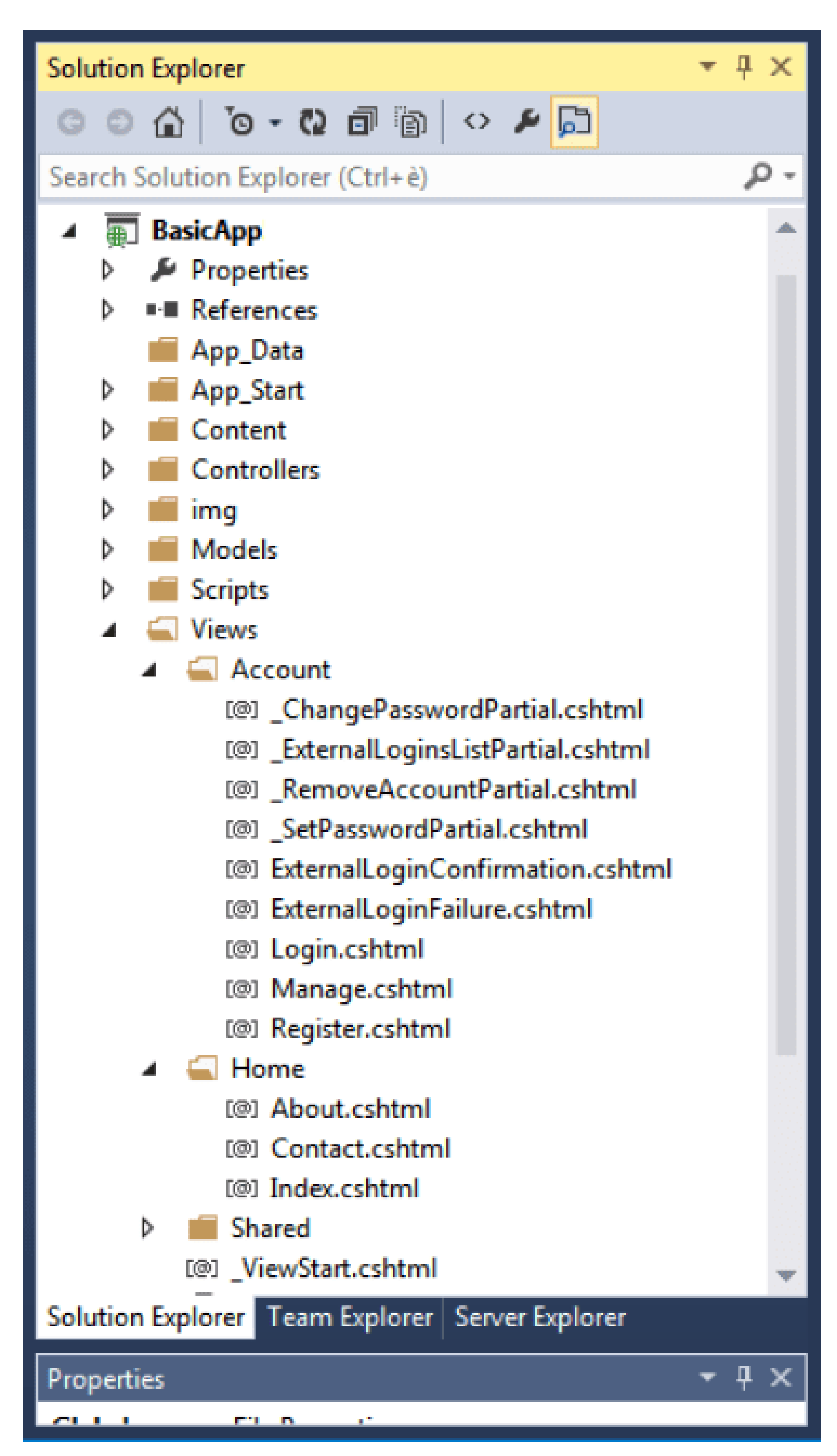


图 2-4　在 ASP.NET MVC 应用程序中定位视图模板

虽然图 2-4 只列出了.cshtml 的 Razor 视图文件，但也支持你混合和匹配按照不同语法所写成的模板文件。

重要提示：

使用不同的标记语言编写视图模板当然不会提高源代码的一致性，但如果团队的成员各自拥有的技能不同，或者当你需要合并一些遗留代码的时候，这就成为了一个可行的解决方案。另外，如果有针对不同引擎的视图模板，解析名称是可以获得一些惊喜的。视图引擎的调用会依照它们的注册顺序，并且在默认情况下，ASPX 引擎会优先于 Razor 引擎。若要修改这一顺序，需要在应用程序启动时清除引擎集合，再重新按你喜欢的顺序添加引擎。

一般情况下，ASP.NET MVC 要求你将每个视图模板放在使用该模版的控制器的文件夹中。如果预期多个控制器会调用同一个视图(或同一视图的一部分)，那么可以把模板文件移到 Shared 文件夹中。

最后，请注意在 Views 文件夹中，在项目级别上存在的同一目录层次结构必须复制到生产服务器上。不过，像 Controller、App_Start 和 ViewModel 这样的文件夹却是普通的名称空间容器，用于更好地组织源文件，所以可以在生产环境中忽略掉。

3. 用于视图的模板

如前所述，视图无非是用来生成 HTML 内容的模板。以下是需要在一些 Views 子文件夹中找到的一个视图模板文件的有效内容。该视图模板面向的是 ASPX 视图引擎。

```
<%@ Page Language="C#"
        MasterPageFile="~/Views/Shared/Site.Master"
        Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="aboutTitle" ContentPlaceHolderID="TitleContent"
runat="server">
    About the book
</asp:Content>

<asp:Content ID="aboutContent" ContentPlaceHolderID="MainContent"
runat="server">
    <h2><%: ViewBag.Message %></h2>
    <p>
        Put content here.
    </p>
</asp:Content>
```

直截了当地说，这看起来与过去可靠的 ASP.NET Web Forms 页面没什么两样。那么 ASP.NET MVC 的关键在哪里？

无可否认，这里你看到的语法与 ASP.NET Web Forms 页面中的确实一样；但是此文件的角色却是完全不同的。在 ASP.NET MVC 中，about.aspx 不是公共资源，因此不可以通过链接或在浏览器的地址栏中输入而向其发出请求。相反，它是一个内部资源文件，用于为视图提供模板。尤其是只有当用户发出一个将系统映射到控制器方法的请求时，about.aspx 模板才会被调用，之后，该控制器方法会选择 About 视图并将其显示出来，如下所示：

```
public ActionResult About()
{
    ViewBag.Message = "Thank you for choosing this book!";
```

```
    // By default, the view name is the same as the action name.
    return View();
}
```

另一个较大的差别是通过该模板所显示的数据。在 Web Forms 中，每个页面都由几个服务器控件组成，而作为开发人员，你要在服务器控件上设置属性以显示数据。在 ASP.NET MVC 中，你把需要传递到视图的数据在一个容器对象中进行分组，再将该容器对象作为选择了该视图的控制器调用中的参数进行传递。

ASP.NET MVC 使你能够在对 View 方法的调用中直接传递容器对象。在任何情况下，都有两个预定义的字典可用于控制器方法填充数据。它们分别是 ViewData 字典和 ViewBag 字典。

需要记住的是，在理想情况下，视图对象不需要自行检索数据；它必须处理的唯一数据只是从控制器所接收的数据。

重要提示：

视图模板的目的是为了生成 HTML，但模板的源并不需要是 HTML。用来编写模板的语言取决于视图引擎。如果选择 ASPX 视图引擎，那么其语言可能与你从 ASP.NET Web Forms 所掌握的 ASPX 标记差不多一样；如果选择 Razor 引擎或其他引擎，则情况会大不一样。

4. 母版视图

与在 ASP.NET Web Forms 中一样，你要决定是从头开始编写视图模板还是从母版视图中继承一些常用的标记。如果选择后者，则需要通过视图引擎定义有关语法和约束设置的母版模板。

指定母版视图是比较容易的。可以使用视图引擎所支持的规则，也可以在从控制器中选择下一个视图时将母版视图的名称作为参数传递给 View 方法。请注意与普通视图相比，母版模板可能会遵循不同的规则。比如，ASPX 视图引擎要求母版模板的扩展名为.master，且需要放在 Shared 文件夹中。而 Razor 视图引擎则要求添加.cshtml 扩展名，并需要你在 Views 文件夹根目录下的一个专用的 viewstart.cshtml 文件中指定路径。

后面还会谈到这两个默认的视图引擎。

2.2　HTML 帮助器

无可否认，ASPX 视图引擎与 ASP.NET Web Forms 非常相似，但两者支持服务器控件的方式却不尽相同。这并非局部实现所造成的，而是有着更深层次的原因。实际上，服务器控件与 Web Forms 页面的生命周期耦合得过于紧密，以至于不能在请求-处理模式中得到相对容易的改造，该请求处理模式会把操作分成清晰的不同阶段，如获取输入数据、处理请求、选

择下一个视图等。

另一方面，服务器控件在 ASP.NET 中提供了一个非常重要的用途——促进 HTML 级别的代码重用。虽然 ASP.NET MVC 非常强调让开发人员获得对每一个 HTML 标记的控制，但很多 HTML 却不能在视图中硬编码。它们需要根据动态发现的数据以编程方式进行构建。在 ASP.NET MVC 中等同于服务器控件的技术是什么？答案就是 HTML 帮助器。

注意：

HTML 帮助器与服务器控件当然不是一回事，但其是在用视图引擎获得 HTML 级代码重用方面最接近于服务器控件的。HTML 帮助器方法没有视图状态、没有回传、也没有页面生命周期和事件。HTML 帮助器就是简单的 HTML 工厂。从技术上讲，HTML 帮助器是在一个系统类——HtmlHelper 类上定义的扩展方法，这个类可以在所提供的输入数据的基础上输出 HTML 字符串。实际上，HTML 帮助器的内部机制就是将文本累加到一个 StringBuilder 对象中。

2.2.1　基础帮助器

ASP.NET MVC 框架提供了一些现成的 HTML 帮助器，包括 CheckBox、ActionLink 和 RenderPartial。表 2-2 提供了一组 HTML 帮助器的列表。

表 2-2　HTML 帮助器方法集

方　法	类　型	描　述
BeginForm, BeginRouteForm	Form	返回一个表示系统用于呈现<form>标记的 HTML 格式的内部对象
EndForm	Form	一个 void 方法，关闭等待的</form>标记
CheckBox, CheckBoxFor	Input	为一个复选框输入元素返回 HTML 字符串
Hidden, HiddenFor	Input	为一个隐藏的输入元素返回 HTML 字符串
Password, PasswordFor	Input	为一个密码输入元素返回 HTML 字符串
RadioButton, RadioButtonFor	Input	为一个单选按钮输入元素返回 HTML 字符串
TextBox, TextBoxFor	Input	为一个文本输入元素返回 HTML 字符串
Label, LabelFor	Label	为一个 HTML 标签元素返回 HTML 字符串
ActionLink, RouteLink	Link	为一个 HTML 链接返回 HTML 字符串
DropDownList, DropDownListFor	List	为一个下拉框列表返回 HTML 字符串
ListBox, ListBoxFor	List	为一个列表框返回 HTML 字符串
TextArea, TextAreaFor	TextArea	为一个文本区域返回 HTML 字符串
Partial	Partial	返回合并到指定用户控件中的 HTML 字符串

(续表)

方　法	类　型	描　述
RenderPartial	Partial	将合并到指定用户控件中的 HTML 字符串写入输出流
ValidationMessage, ValidationMessageFor	Validation	为一个验证消息返回 HTML 字符串
ValidationSummary	Validation	为一个验证摘要消息返回 HTML 字符串

举个例子，让我们看看如何使用 HTML 帮助器来创建一个以编程方式确定文本的文本框。如果使用 ASPX 视图引擎，则可以在一个代码块中放置下面的调用：

```
<%: Html.TextBox("TextBox1", ViewBag.DefaultText) %>
```

或者，如果使用 Razor 视图引擎，则需要在该调用前面添加@符号(将很快在本书的其他部分谈到 Razor)。

```
@Html.TextBox("TextBox1", ViewBag.DefaultText)
```

注意：

该代码片段中的 Html 涉及基类的内置属性，两个视图引擎都会使用它来引用呈现的视图。该类对于 ASPX 视图引擎来说是 ViewPage，而对于 Razor 视图引擎则是 WebPage。在这两种情况下，属性 Html 都是 HtmlHelper 的一个实例。

每个 HTML 帮助器均有一大堆重载来让你指定特性值和其他相关信息。比如，以下是如何通过使用类特性来设置文本框的风格：

```
<%: Html.TextBox("TextBox1",
                 ViewBag.DefaultText,
                 new Dictionary<String, Object>{{"class", "coolTextBox"}}) %>
```

在表 2-2 中你能看到很多形如 xxxFor 的帮助器。它们与其他帮助器相比有哪些不同呢？一个形如 xxxFor 的帮助器与基础版本的不同之处在于它只接受 lambda 表达式，比如下面的这个帮助器：

```
<%: Html.TextBoxFor(model => model.FirstName,
                    new Dictionary<String, Object>{{"class", "coolTextBox"}}) %>
```

对于文本框，lambda 表达式会指出要在输入字段中显示的文本。当要填充视图的数据在一个模型对象中进行分组时，xxxFor 变体尤其有用。在这种情况下，视图结果的读取会更清晰，并且是强类型化的。

让我们看看基本 HTML 帮助器的其他一些例子。

1. 呈现 HTML 表单

在 ASP.NET MVC 中呈现表单，最让人厌烦的事情是需要指定目标 URL。BeginForm 和 BeginRouteForm 帮助器能承担这一令人不快的工作。下面的代码片段显示了如何编写一个呈现用户和密码文本框的简单输入表单：

```
<% using (Html.BeginForm()) { %>
  <div>
    <fieldset>
        <legend>Account Information</legend>
        <p>
            <label for="userName">User name:</label>
            <%= Html.TextBox("userName") %>
            <%= Html.ValidationMessage("userName") %>
        </p>
        <p>
             <label for="password">Password:</label>
             <%= Html.Password("password") %>
             <%= Html.ValidationMessage("password") %>
        </p>
        ...
        <p>
            <input type="submit" value="Change Password" />
        </p>
    </fieldset>
  </div>
<% } %>
```

BeginForm 帮助器负责<form>开始标记。但 BeginForm 方法并不直接发出任何标记。它仅限于创建 MvcForm 类的实例，该实例之后会添加到页面的控件树并随后呈现。

默认情况下，BeginForm 会呈现一个回发到相同 URL 并随后回发至相同控制器操作的表单。使用 BeginForm 方法的其他重载，可以指定目标控制器的名称和操作、该操作的任意路由值、HTML 特性，甚至可以指定是否需要表单执行 GET 或 POST 动作。BeginRouteForm 的作用与 BeginForm 差不多，但有一点不同，它可以从任意一组路由参数开始而生成一个 URL。换句话说，BeginRouteForm 并不局限于基于控制器名称和操作的默认路由。

我相信探讨 HTML 表单是展示 Razor 的附加值优于 ASPX 的最佳方式。下面显示了如何使用 Razor 语法重写先前的表单：

```
@using (Html.BeginForm())
{
```

```
    <div>
      <fieldset>
         <legend>Account Information</legend>
         <p>
            <label for="userName">User name:</label>
            @Html.TextBox("userName")
            @Html.ValidationMessage("userName")
         </p>
         <p>
             <label for="password">Password:</label>
             @Html.Password("password")
             @Html.ValidationMessage("password")
         </p>
         ...
         <p>
            <input type="submit" value="Change Password" />
         </p>
      </fieldset>
    </div>
}
```

本书的其余部分都会将 Razor 用于视图。

注意：

在 HTML 中，当你使用表单标记时，只能使用 GET 和 POST 动词提交一些内容。而在 ASP.NET MVC 中，HtmlHelper 类上的一个原生方法——HttpMethodOverride——解决了这个问题。该方法会发出一个隐藏字段，其名称硬编码到 X-HTTP-Method-Override，并且其值是 PUT、DELETE 或 HEAD。该隐藏字段的内容会重载表单的方法集，因此你也能够从浏览器的内部调用 REST API 了。你还可以用相同的 X-HTTP-Method-Override 名称指定 HTTP 标头中的重载值，或者将查询字符串值中的重载值指定为名称/值对。重载只对 POST 请求有效。

2. 呈现输入元素

你能在表单中使用的所有 HTML 元素都有一个 HTML 帮助器来加速开发。再次说明一下，从功能的角度来看使用帮助器与使用普通的 HTML 之间并无差别。下面是一个复选框元素的示例，初始设置为 true 但禁用：

```
@Html.CheckBox("ProductDiscontinued",
               true,
               new Dictionary<String, Object>() {{"disabled", "disabled"}}))
```

你还拥有将验证消息与输入字段相关联的工具。如果指定的字段中包含错误，则可以使

用 Html.ValidationMessage 帮助器来显示验证消息。该消息可以通过帮助器中的一个附加参数显式指示出来。所有的验证消息之后会通过 Html.ValidationSummary 帮助器聚合与显示出来。

你会在第 4 章“输入表单”中看到对输入表单和验证的进一步探讨。

3. 操作链接

如前文所述，以编程方式创建 URL 是 ASP.NET MVC 中一个乏味且易出错的任务。因此帮助器大受欢迎，特别是针对这种任务。事实上，ActionLink 帮助器是 ASP.NET MVC 视图中最常用到的一个。下面是一个示例：

```
@Html.ActionLink("Home", "Index", "Home")
```

通常情况下，操作链接需要链接文本、操作名称和可选的控制器名称。该示例产生的 HTML 如下：

```
<a href="/Home/Index">Home</a>
```

此外，可以指定路由值、锚点标记的 HTML 特性、甚至是一个协议(比如 HTTPS)、主机和片段。

RouteLink 帮助器的工作方式差不多相同，但它不需要你来指定操作。有了 RouteLink 帮助器，就可以使用任何已注册的路由名称来确定生成的 URL 的模式了。

由 ActionLink 发出的文本会进行自动编码。这就是说你不能在链接文本中使用任何可能导致浏览器将其看作 HTML 的 HTML 标记。尤其不能把 ActionLink 用作视图按钮和图像链接。不过，若要生成基于控制器和操作数据的链接，则可以使用 UrlHelper 类。

UrlHelper 类的实例与 ViewPage 类型上的 Url 属性相关联。下面的代码显示了实际使用中的 Url 对象:

```
<a href="@Url.Action("Edit")">
    <img src="editMemo.jpg" alt="Edit memo" />
</a>
```

UrlHelper 类有几个方法的作用类似于 ActionLink 和 RouteLink。它们的名称是 Action 和 RouteUrl。

注意：

ASP.NET MVC 路由不能识别子域；它总是假定你处于应用程序路径之中。这意味着如果想要在单个应用程序内使用子域，而不是在虚拟路径(例如，使用 dino.blogs.com 而不是 www.blogs.com/dino)中，那么找出你位于哪个子应用程序的额外工作就完全是你自己的事情了。有多种方式可以解决这个问题。一个简单的方法是创建一个自定义路由处理程序，由它查看在 URL 中传递的主机，并决定设置哪个控制器。但是这个解决方案仅限于固定的传入请

求。它可能并不能满足用于生成到资源和操作的链接的所有帮助器。一个更完整的解决方案是创建一个能识别子域的 Route 类。可以在 http://blog.maartenballiauw.be/post/2009/05/20/ASPNET-MVC-Domain-Routing.aspx 找到一个合适的例子。

4. 部分视图

可以使用 Partial 或 RenderPartial 帮助器方法来插入一个部分视图。这两种方法都采用该部分视图的名称作为参数。两者的唯一区别是 Partial 返回一个字符串，而 RenderPartial 会写入到输出流，并返回空。因此，两者的用法稍有不同，如下所示：

```
@Html.Partial("login")
@Html.RenderPartial("login")
```

在 ASP.NET MVC 中，部分视图类似于 Web Forms 中的用户控件。部分视图的典型位置位于 Views 中的 Shared 文件夹。但是也可以将部分视图存储在控制器专用的文件夹中。部分视图包含在视图中，但会被视作完全独立的实体。实际上，为一个视图引擎编写一个视图而该视图的部分视图却需要另一个视图引擎的情形是完全合理的。

5. HtmlHelper 类

HtmlHelper 类受欢迎的原因很大程度上归功于其众多的扩展方法，但它也有不少有用的原生方法。表 2-3 列出了其中的一些。

表 2-3　HtmlHelper 最受欢迎的原生方法

方　法	描　述
AntiForgeryToken	为存储了防伪令牌的隐藏输入字段返回 HTML 字符串(更多详细信息请参见第 4 章)
AttributeEncode	使用 HTML 编码规则对指定特性值进行编码
EnableUnobtrusiveJavaScript	以隐式方式设置允许帮助器生成 JavaScript 代码的内部标识
EnableClientValidation	设置允许帮助器生成用于客户端验证的代码的内部标识
Encode	使用 HTML 编码规则对指定值进行编码
HttpMethodOverride	为用于重写有效 HTTP 动词以表明所需的 PUT 或 DELETE 操作的隐藏输入字段返回 HTML 字符串
Raw	返回未编码的原始 HTML 字符串

此外，HtmlHelper 类还提供了大量的公共方法，这些方法在视图内部几乎没有用处，却为编写自定义 HTML 帮助器方法的开发人员提供了大力支持。GenerateRouteLink 就是一个好例证，它会返回一个包含指定路由值虚拟路径的锚点标记。

2.2.2　模板化帮助器

一遍遍地编写 HTML 模板是非常乏味枯燥的，并且容易导致代码出错。模板化帮助器就能帮上大忙，因为它们被创建来使用 C#类的实例、读取属性、并决定如何最好地呈现这些值。通过使用特殊特性来设置视图模型对象，你便为帮助器提供了关于用户界面提示和验证的进一步指示。

有了模板化帮助器，就不会失去对用户界面的控制了；简言之，模型中的特性建立起一系列的规则，将你从繁重的重复劳动中解放了出来。

1. 模板化帮助器的多种类型

在 ASP.NET MVC 中，你有两个基本的模板化帮助器：Editor 和 Display。它们相互协作，使代码在标识、显示和编辑数据对象方面更易于编写和维护。使用这些帮助器的最优方案是在带注释的对象附近编写你的列表或输入表单。不过，模板化帮助器可以同时作用于标量值和复合对象。

模板化帮助器其实具有三个重载。以 Display 帮助器为例，它有以下三个更具体的帮助器：Display、DisplayFor 和 DisplayForModel。这三者之间并无功能上的差异。它们的唯一区别在于各自能够处理的输入参数。

2. Display 帮助器

Display 帮助器接受一个在 ViewData 字典中或视图模型对象上指明属性名称的待处理字符串：

```
@Html.Display("FirstName")
```

DisplayFor 帮助器接受一个 lambda 表达式，并要求把视图模型对象传递给视图：

```
@Html.DisplayFor(model => model.FirstName)
```

最后，DisplayForModel 是获取 model => model 表达式的 DisplayFor 的快捷方式：

```
@Html.DisplayForModel()
```

各种类型的模板化帮助器都有处理元数据(如果有的话)的特殊能力，并可根据情况对其呈现做相应的调整；比如显示标签和添加验证。可以通过使用模板自定义显示和编辑功能，这在后面还会谈到。使用自定义模板的能力适用于所有类型的模板化帮助器。

重要提示：

ViewBag 是在被定义为动态类型的 ControllerBase 类上定义的属性。在.NET 中，动态类型表示动态解释代码的网站。换句话说，每当编译器满足了对动态对象的引用时，它就会发出一

段代码，在运行时检查其是否可以解析和执行。从功能上讲，这类似于 JavaScript 对象的情况。

lambda 表达式不支持动态成员，因此不能用于将数据传递到 ViewBag 字典。另外需要注意的是，为了成功使用 HTML 帮助器中的 ViewBag 内容，你必须将表达式转换成有效类型。

3. Editor 帮助器

Editor 帮助器的目的是让你编辑指定的值或对象。该编辑器会识别它所获取的值的类型，并挑选量身定做的模板进行编辑。预定义的模板有用于对象、字符串、布尔值和多行文本的，而数字、日期和全局统一标识符(GUID)则会退回到字符串编辑器。Editor 帮助器极其擅长处理复杂类型。它通常会遍历每一个公共属性，并为子值建立一个标签和编辑器。下面是在编辑器中，如何在一些传递到视图的对象上显示 FirstName 属性的值：

```
@Html.EditorFor(person => person.FirstName)
```

可以通过在视图的 EditorTemplates 文件夹中创建几个部分视图来自定义编辑器(和可视化工具)。它可以位于控制器专有的子文件夹中，也可以位于 Views\Shared 文件夹中。对于 ASPX 视图引擎，部分视图表现为一个.ascx 模板，而如果正使用 Razor 视图引擎，则部分视图表现为一个.cshtml 模板。可以为每一个所期望支持的类型提供一个自定义模板。例如，在图 2-5 中可以看到一个 datetime.cshtml 模板，它将用于修改日期在编辑与显示中的呈现方式。同样地，可以为视图模型对象中每个类型的属性提供一个部分视图。

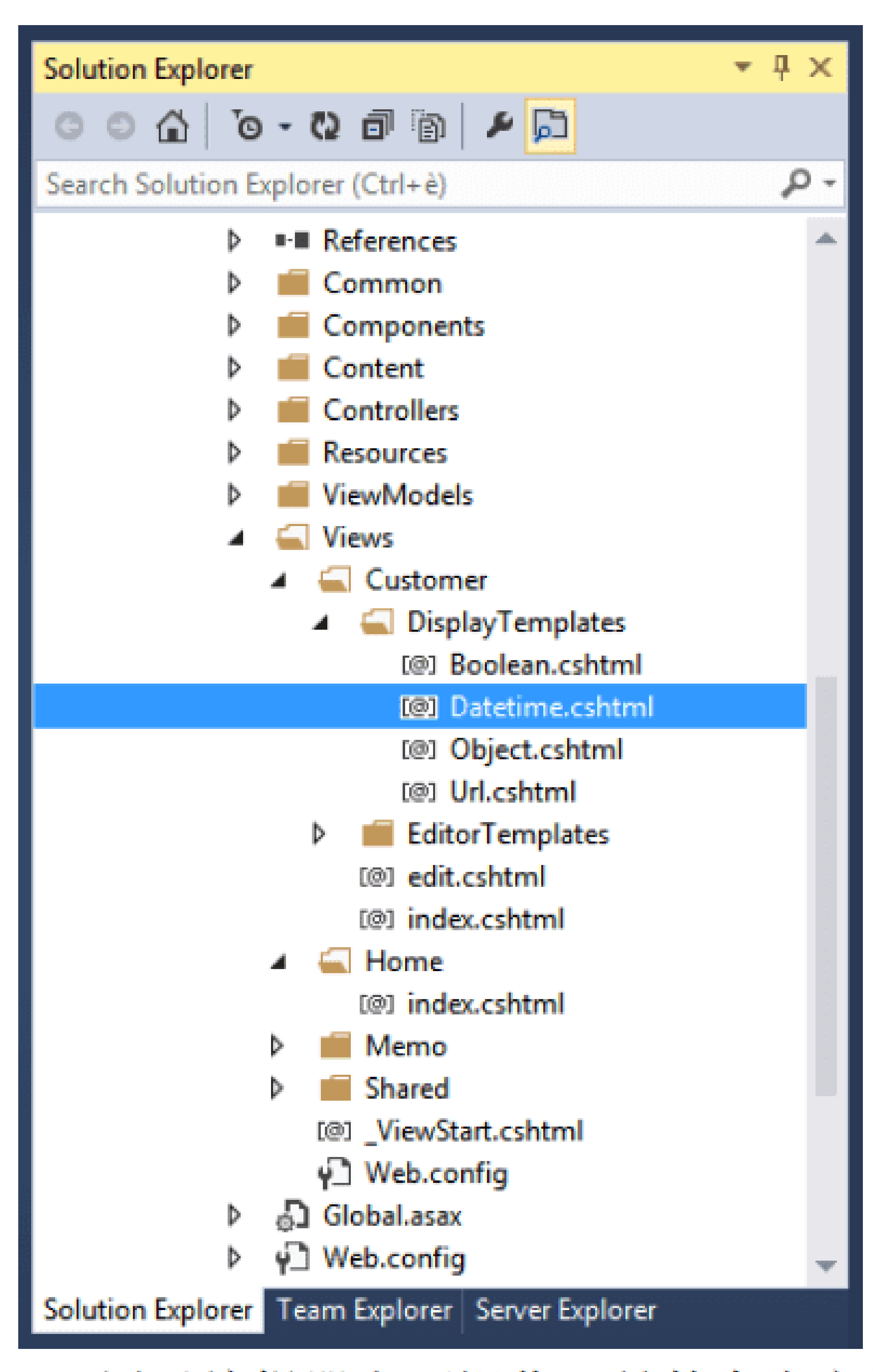


图 2-5　用于编辑器和可视化工具的自定义模板

如果部分视图的名称与类型名称相匹配，则系统会自动选取自定义模板。

也可以通过名称将编辑器指向你的模板，并为模板取一个你喜欢的名称。下面是一个使用 date.ascx 视图来编辑 DateTime 属性的示例：

```
@Html.EditorFor(person => person.Birthdate, "date")
```

在同一个 ASP.NET MVC 应用程序中，可以拥有请求不同的视图引擎的视图。请注意 ASP.NET MVC 会自主解析每一个视图和部分视图。这意味着如果正在运行，比如 about 视图的过程，那么该过程会最终结束于 Razor 引擎(如图 2-5 所示)。但是，如果 about 视图需要一个用于日期的编辑器而你有一个匹配的.aspx 模板，那么不用你提供类似的.cshtml 模板，about 视图也会被选取出来的。

最后，Editor 帮助器可以识别视图模型对象上的数据批注特性，并使用这些信息来添加特殊的验证功能，比如确保给定值落在指定的范围内或不为空。

注意：

当你使用 DisplayForModel 和 EditorForModel 时，系统会使用反射来查找指定对象的所有属性，然后为每个属性生成一个标签和可视化工具或者编辑器。生成视图的整体模板是标签和可视化工具/编辑器的垂直序列。发出的每个 HTML 段均会绑定到一个 CSS 类，并且可以轻松地设置成你喜欢的样式。此外，如果想更改视图模板，则需要提供一个 object.aspx(或 object.cshtml)模板，还要使用反射。在第 4 章中还会谈到这个问题，并举出相关示例。

2.2.3　自定义帮助器

HTML 帮助器方法的原生集合绝对大有裨益，但它对于许多实际的应用程序可能是不合适的。原生帮助器实际上只涵盖了基本的 HTML 元素标记。在这方面，HTML 帮助器与服务器控件显著不同，因为它们完全缺乏对 HTML 的抽象化。但是 HTM 帮助器集的扩展却是很容易的。如果有意创建一个处理 Ajax 任务的 HTML 工厂，那么全部所需只是 HtmlHelper 类或 AjaxHelper 类的一个扩展方法。

1. HTML 帮助器的结构

HTML 帮助器是一个普通的方法，不依赖于任何强制的原型模式。通常你要设计 HTML 帮助器方法的签名，使它只接受需要的数据。有几个重载或可选参数也是常见现象。

从内部来看，帮助器处理输入数据并首先会通过在缓存区累加文本来构建输出 HTML。这是最灵活的方法，但当应用的逻辑变得复杂的时候，也是难于管理的。另一种方法包含使用 TagBuilder 类来生成 HTML，该类提供了面向 HTML 的字符串生成器工具。TagBuilder 类为你生成 HTML 标记的文本，从而可以使你通过串联标签而非普通的字符串以构建大块的 HTML。

HTML 帮助器预期会返回一个已编码的 HTML 字符串。

2. MvcHtmlString 不只是一个字符串

如果对一个已编码的标记片段使用紧凑语法会怎么样？如果缺乏应对措施，文本就会不可避免地被双重编码。因此，最好的做法是编写可以返回 MvcHtmlString 封装对象而非普通字符串的 HTML 帮助器。实际上，所有的原生 HTML 帮助器都会被重构以返回 MvcHtmlString。这一更改对开发人员来说不算什么。通过下面的代码，可以很容易地从一个字符串中获得 MvcHtmlString 对象：

```
var html = GenerateHtmlAsString();
return MvcHtmlString.Create(html);
```

MvcHtmlString 类型是对包含 HTML 内容的字符串的一个智能封装，它公开了 IHtmlString 接口。

IHtmlString 的目的是什么？在 ASP.NET 中试图对实现 IHtmlString 的对象进行 HTML 编码会造成空操作指令。

3. HTML 帮助器样例

假定你的视图接收了一些可以为空的文本。不过你并不想呈现空的字符串；而宁愿显示一些默认的文本，如 N/A。要如何做才行？只用一个 if 语句就可以完美地解决一切。但是在 ASPX 标记中嵌套 if 语句并不会有助于代码的简洁；在 Razor 中情况也好不到哪里去。

一个专用帮助器可以摆平一切，因为它封装了 if 语句，并且提供更加紧凑和易读的代码，同时还保留了所需的逻辑。下面的代码演示了一个 HTML 帮助器，如果给定的字符串为 null 或空的话，它会用一些默认文本替换该字符串：

```
public static class OptionalTextHelpers
{
   public static MvcHtmlString OptionalText(this HtmlHelper helper,
         String text,
         String format="{0}",
         String alternateText="",
         String alternateFormat="{0}")
   {
         var actualText = text;
         var actualFormat = format;

         if (String.IsNullOrEmpty(actualText))
         {
            actualText = alternateText;
            actualFormat = alternateFormat;
         }
```

```
            return MvcHtmlString.Create(String.Format(actualFormat, actualText));
        }
    }
```

帮助器最多可以有四个参数，其中三个为可选参数。帮助器用到了原始文本及其 null 的替换，另外还有一个格式化字符串来修饰这两种情况下的文本。

4. Ajax 帮助器样例

Ajax 帮助器与 HTML 帮助器的区别仅仅由于它是在 Ajax 操作的上下文中调用的。例如，我们假设你需要将某个图片用作按钮。单击该图片应该自动触发对某些应用程序 URL 的 Ajax 调用。

那么这与只是将一些 JavaScript 附加到图片的 click 事件然后使用 jQuery 进行调用有什么不同呢？如果知道传递到 jQuery 的 URL，则不需要此帮助器。但是如果发现将 URL 表达为控制器/操作对更好的话，就需要这个帮助器来生成一个链接，把用户带到控制器/操作对所指向的任何地方，如下所示：

```
public static class AjaxHelpers
{
    public static String ImgActionLink(this AjaxHelper ajaxHelper,
        String imageUrl,
        String imgAltText,
        String imgStyle,
        String actionName,
        String controllerName,
        Object routeValues,
        AjaxOptions ajaxOptions,
        Object htmlAttributes)
    {
        const String tag = "[xxx]";    // arbitrary string
        var markup = ajaxHelper.ActionLink(
             tag, actionName, controllerName, routeValues, ajaxOptions,
             htmlAttributes).ToString();

        // Replace text with IMG markup
        var urlHelper = new UrlHelper(ajaxHelper.
                                          ViewContext.RequestContext);
        var img = String.Format(
            "<img src='{0}' alt='{1}' title='{1}' style='{2}' />",
            urlHelper.Content(imageUrl),
            imgAltText,
            imgStyle);
        var modifiedMarkup = markup.Replace(tag, img);
```

```
            return modifiedMarkup;
        }
    }
```

该帮助器首先会调用默认的 ActionLink 帮助器来获取 URL，正如它就是一个基于文本的超链接一样。第一步要把超链接文本设置为一个已知字符串，用作占位符。接下来，当一切准备好以后，该帮助器会移除占位符字符串，替换为图片的 URL。

为什么不能直接提供<img>标记用作原始操作链接的文本呢？ActionLink 很好地遵循其规则，它会对所有内容进行 HTML 编码，因此你看不到任何图片，只有 URL 的文本。

2.3　Razor 视图引擎

一方面，Web Forms 视图引擎采用了一种几乎对所有 ASP.NET 开发人员都很熟悉的语法。但是在另一方面，ASPX 标记被设计用作支持服务器控件的一种方式，当主要与代码块一起使用时，会表现出严重的局限性。很明显，其主要问题是缺乏可读性。

在 ASP.NET MVC 3 中引入的 Razor 视图引擎解决了这个问题，它提供了一种替代标记语言来定义视图模板的结构。

2.3.1　视图引擎的内部机制

根据 Razor，视图模板就是一个 HTML 页面加上几个占位符和代码片段。总体来说，视图模板的可读性大大提高了，并且通过将 Razor 代码片段与 HTML 帮助器相结合，可以整理视图，使它们更易于阅读和维护。

1. 搜索位置

Razor 视图引擎支持一些与视图模板的位置有关的有趣属性。表 2-4 描述了这些属性。

表 2-4　表示用于视图模板所需位置格式的属性

属　性	描　述
AreaMasterLocationFormats	在以区域为基础的应用程序中搜索到的母版视图的位置
AreaPartialViewLocationFormats	在以区域为基础的应用程序中搜索到的部分视图的位置
AreaViewLocationFormats	在以区域为基础的应用程序中搜索到的视图的位置
MasterLocationFormats	搜索到的母版视图的位置
PartialViewLocationFormats	搜索到的部分视图的位置
ViewLocationFormats	搜索到的视图的位置
FileExtensions	视图、部分视图和母版视图的扩展名支持列表

每个属性都是由一个字符串数组实现的。表 2-5 显示了这些属性的默认值。

表 2-5　默认位置格式

属　　性	默认位置格式
AreaMasterLocationFormats	~/Areas/{2}/Views/{1}/{0}.cshtml ~/Areas/{2}/Views/Shared/{0}.cshtml ~/Areas/{2}/Views/{1}/{0}.vbhtml ~/Areas/{2}/Views/Shared/{0}.vbhtml
AreaPartialViewLocationFormats	~/Areas/{2}/Views/{1}/{0}.cshtml ~/Areas/{2}/Views/{1}/{0}.vbhtml ~/Areas/{2}/Views/Shared/{0}.cshtml ~/Areas/{2}/Views/Shared/{0}.vbhtml
AreaViewLocationFormats	~/Areas/{2}/Views/{1}/{0}.cshtml ~/Areas/{2}/Views/{1}/{0}.vbhtml ~/Areas/{2}/Views/Shared/{0}.cshtml ~/Areas/{2}/Views/Shared/{0}.vbhtml
MasterLocationFormats	~/Views/{1}/{0}.cshtml ~/Views/Shared/{0}.cshtml ~/Views/{1}/{0}.vbhtml ~/Views/Shared/{0}.vbhtml
PartialViewLocationFormats	~/Views/{1}/{0}.cshtml ~/Views/{1}/{0}.vbhtml ~/Views/Shared/{0}.cshtml ~/Views/Shared/{0}.vbhtml
ViewLocationFormats	~/Views/{1}/{0}.cshtml ~/Views/{1}/{0}.vbhtml ~/Views/Shared/{0}.cshtml ~/Views/Shared/{0}.vbhtml
FileExtensions	.cshtml, .vbhtml

差不多所有属性的使用方式都与 Web Forms 引擎的相同。对于视图而言，唯一的变化是不同的文件扩展名和不同的语法。让我们来熟悉一下吧。

2. 代码碎块

Razor 视图模板实质上是一个 HTML 页面加上几个代码段，也称为代码碎块。代码碎块类似于 ASP.NET 代码块，但它们的特点是语法更简洁明了。Razor 代码块的标志是开头使用@字符。更为重要的是，你不需要显式关闭这些代码块。Razor 分析器使用 Visual Basic 或 C#解析逻辑来找出代码行的结束位置。下面有一个示例：

```
<html>
<head>
   <title>@ViewBag.Title</title>
   <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet"
              type="text/css" />
   <script src="@Url.Content("~/Scripts/jquery.js")"
                type="text/javascript"></script>
</head>
...
</html>
```

可以将任何 Razor 代码碎块与普通的标记相混合，即使代码碎块中包含像 if/else 或 for/foreach 语句这样的控制流语句。下面的代码利用代码碎块编写了一个页面：

```
<body>
   <h2>@ViewBag.Header</h2>
   <hr />

   <table>
      <thead>
        <th>City</th>
        <th>Country</th>
        <th>Been there?</th>
      </thead>
   @foreach (var city in Model) {
     <tr>
       <td><%: city.Name %></td>
       <td><%: city.Country %></td>
       <td><%: city.Visited ?"Yes" :"No"%></td>
     </tr>
   }
   </table>
</body>
```

注意，位于源代码中间的右大括弧(})，它通过解析器得到正确的识别和解释。

一般来说，可以在 Razor 模板中使用任何 C#(或 Visual Basic)指令，只要给它们加上前缀

@。下面这个示例展示了如何引用一个名称空间以及创建一个表单块：

```
@using YourApp.Extensions;
...
<body>
   @using (Html.BeginForm()) {
      <fieldset>
          <div class="editor-field">
               @Html.TextBox("TextBox1")
          </div>
      </fieldset>
   }
</body>
```

归根结底，Razor 模板就是带有封装了可执行语句和 HTML 帮助器的@表达式的普通 HTML 标记文件。不过，存在几个快捷方式罢了。

3. 代码碎块的专用表达式

可以通过在@{code}块中进行封装来在任意位置插入整段的多行代码，就像下面所示的这个：

```
@{
   var user = "Dino";
}
...
<p>@user</p>
```

可以检索自己创建的任意变量，并且之后可以像属于单个代码块的代码一样使用它。@{...}块的内容可以混合代码和标记。不过，重要的是解析器可以找出代码于何处结束，标记在何处开始，反之亦然。请看看下面的代码碎块：

```
@{
    var number = GetRandomNumber();
    if(number.IsEven())
      <p>Number is even</p>
    else
      <text>Number is odd</text>
}
```

如果发出的标记内容被 HTML 标签封装起来了，则解析器会正确地将其作为标记识别出来。如果是普通文本(比如一个文本字符串)，就需要将其用 Razor 专用的<text>封装起来，以便解析器能够正确地处理它。

来源于表达式的单个语句可以通过使用括号在同一表达式中组合：

```
<p> @("Welcome, " + user) </p>
```

在你需要放置一个函数调用时，括号也起作用：

```
<p> @(YourMethod(1,2,3)) </p>
```

由 Razor 进行处理的任何内容都是自动编码的，所以不需要你处理。如果的代码返回 HTML 标记，而你希望原样发出，不进行自动编码，那么你应该采用如下所示的 Html.Raw 帮助器方法：

```
@Html.Raw(Strings.HtmlMessage)
```

最后，在多行代码碎块@{...}的内部，你要使用 C#或 Visual Basic 语言的语法来放置注释。可以通过使用@*...* @语法来注释整个 Razor 代码块，如下所示：

```
@*
<div> Some Razor markup </div>
*@
```

用于添加或移除注释块的 Visual Studio 工具栏按钮可以很好地支持 Razor 语法。

注意：

大部分时候，Razor 解析器足以智能地从上下文中获悉你使用@符号的原因，无论是表示代码碎块还是电子邮件文本地址。如果遇到解析器不能成功解析的极端情况，请使用@@明确表示出你希望@符号按文本解析，而不是代码碎块的起始符。

4. 条件式代码碎块

Razor 还支持一些条件特性，只有在 Razor 表达式为真或不为空的时候才应发出。让我们思考下面的标记：

```
<div class="@yourCss">

   ...

</div>
```

变量 yourCss 可能为空或 null；在这种情况下理论上你不会想要发出这一类特性。在 ASP.NET MVC 4 以前，你不得不自己写一些条件逻辑。而现在，条件逻辑已经内置在 Razor 引擎中了。

请注意如果 Razor 表达式是布尔值，情况是一样的：如果它返回 false 或者为 null，该特

性不应发出。

5. Razor 视图对象

Razor 视图引擎在使用时，生成的视图对象是一个在 System.Web.Mvc 程序集中定义的 WebViewPage 类的实例。这个类集成了解析标记和呈现 HTML 的逻辑。该类上的公共属性对于你在实际模板中编写的任何代码碎片都可用。

表 2-6 提供了你可能感兴趣的几个属性的简单列表。

表 2-6　Razor 视图对象的常用属性和方法

属　　性	描　　述
Ajax	获取一个用于在模板中引用 Ajax HTML 帮助器的 AjaxHelper 类的实例
Culture	获取和设置与当前请求相关的区域性 ID。该设置会影响页面区域性依赖特性，比如日期、数字和货币格式。区域性表现为 xx-yy 格式，其中 xx 表示语言、yy 表示区域
Href	将你在服务器端代码(其中可以包含~运算符)中创建的路径转换成浏览器理解的路径
Html	获取一个用于在模板中引用 HTML 帮助器的 HtmlHelper 类的实例
Context	获取中心资源库以获得各种 ASP.NET 内部对象的访问权：请求、响应、服务器、用户等等
IsAjax	如果当前请求由浏览器的 Ajax 对象初始化，则返回 true
IsPost	如果当前请求通过 HTTP POST 动词发出，则返回 true
Layout	获取和设置包含母版视图模板的文件路径
Model	获取一个包含视图数据的视图模型对象(如果有的话)的引用。这一属性是动态类型
UICulture	获取和设置与当前请求相关的用户界面区域性 ID。这一设置会判定在多语言应用程序中加载哪个资源。区域性表现为 xx-yy 格式，其中 xx 表示语言、yy 表示区域
ViewBag	获取一个 ViewBag 字典的引用，该字典可能包含控制器需要传递给视图对象的数据
ViewData	获取一个 ViewData 字典的引用，该字典可能包含控制器需要传递给视图对象的数据

并不是所有这些属性都能直接在 WebViewPage 类上得到有效的定义。其中很多实际上是在父类上定义的。

2.3.2　设计一个样例视图

要创建一个具有一定复杂度的 Razor 视图，需要了解如何将数据传递给视图，以及如何定义母版视图。

1. 定义用于视图的模型

如前所述，控制器能以各种方式将数据传递给视图。它可以使用全局字典，如 ViewBag 或 ViewData。更好的是，控制器可以使用为特定视图量身定做的强类型对象。稍后我会详述各种方法的利弊。

要从代码碎块中使用 ViewBag 或 ViewData，不需要采取任何特别措施。只需要编写读入或写进字典的@表达式即可。相反，要使用强类型的视图模型，你需要在模板文件的顶部声明实际的类型，如下所示：

```
@model IList<GridDemo.Models.City>
```

用于表示类型的语法与你在整个模板中编写代码碎块所使用的语言相同。接着，通过使用 Model 属性访问视图模型对象中的属性，如表 2-6 中所列示的。下面是一个示例：

```
@model IList<GridDemo.Models.City>
@{
   ViewBag.Title = ViewBag.Header;
}

<h2>@ViewBag.Header</h2>
<hr />

<table>
    <thead> ... </thead>
@foreach (var city in Model)
{
    <tr> ...  </tr>
}
</table>
```

当然，如果 Model 引用了具有子属性的对象，那么可以使用 Model.Xxx 来引用每个子属性。

注意：

在基于 Visual Basic 的 Razor 视图中，定义视图模型对象是通过使用不同的语法来实现的。不是使用@model 关键字，而要用@ModelType 关键字。

2. 定义母版视图

在 Razor 中，布局页面扮演着 Web Forms 中母版页的角色。布局页面是一个标准的 Razor 模板，视图引擎会将你定义的任意视图呈现出来，从而将外观和感觉与网站版面统一起来。

每个视图都可以通过简单地设置布局属性来定义其布局页面。布局可以设置成硬编码文

件或设置成从计算运行时条件中所产生的任意路径：

```
@{
    if (Request.Browser.IsMobileDevice) {
        Layout = "mobile.cshtml";
    }
}
```

你不必在每个视图文件中都集成确定布局的代码。使用 Razor，可以在 Views 文件夹中定义一个特殊文件，该文件在每个视图构建和呈现之前进行处理。这个文件被称为_ViewStart.cshtml，是任何与视图相关的启动代码的理想容器，其中包括决定使用哪个布局的代码。下面是_ViewStart.cshtml 文件的一个常见实现：

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

根据前面的代码片段，_Layout.cshtml 文件定义了网站中每个视图的总体结构(这只是与母版页对应的 Razor)。

布局页面包含了 HTML 代码和代码碎块的常见组合，是从 WebViewPage 派生而来的。因此，它可以访问 WebViewPage 的任何属性，包括 ViewBag 和 ViewData。布局模板必须包含至少一个占位符用于注入特定视图的代码。占位符通过调用 RenderBody 方法(在 WebViewPage 上定义)来表达，如下面这个示例所揭示的：

```
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet"
               type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.9.1.min.js")"
                type="text/javascript"></script>
</head>
<body>
    <div class="page">
    @RenderBody()
   </div>
</body>
</html>
```

执行 RenderBody 方法会导致实际视图中的任何代码都注入布局模板。实际视图模板中的代码会在视图和布局合并之前进行处理。这意味着可以在布局能够检索和使用的 ViewBag 中编写以编程方式设置值的视图代码。一个典型例子就是网页标题。另一个适用示例是

<meta>标签。

重要提示：

在视图中，建议你通过使用波浪号(~)运算符将诸如图片、脚本和样式表等资源引用到网站的根目录。在 ASP.NET MVC 的早期版本中，你不得不借助 Url.Content 方法，以确保波浪号(~)在 URL 中的正确扩展。从 ASP.NET MVC 4 开始，波浪号(~)通过 Razor 引擎自动扩展，虽然 Url.Content 的使用依然受到支持，但已不再必要。

3. 定义节

RenderBody 方法定义了在布局内注入的单个点。虽然这是一种常见情形，但你可能需要将内容注入到多个位置。在布局模板中，你通过在希望节所出现的位置放置一个对 RenderSection 的调用来定义注入点：

```
<body>
    <div class="page">
    @RenderBody()
   </div>
   <div id="footer">
      @RenderSection("footer")
   </div>
</body>
```

每一个节都由名称标识，并能标记为可选。要声明某个节是可选的，可以参考下面的代码：

```
<div id="footer">
   @RenderSection("footer", false)
</div>
```

RenderSection 方法接受可选的布尔值参数，它表明是否需要某一节。下面的代码在功能上等同于之前的代码，但从可读性的角度来说要好得多：

```
<div id="footer">
   @RenderSection("footer", required:false)
</div>
```

注意，那个 required 并不是关键字；简言之，它是在 RenderSection 方法上定义的形式参数的名称(其名称的友好显示得益于智能提示)。

如果视图模板不包含所需的节，你会得到一个运行时异常。下面展示了如何在视图模板中定义节的内容：

```
@section footer {
```

```
    <p>Written by Dino Esposito</p>
}
```

可以在 Razor 视图模板的任何位置定义节的内容。

如果需要将整个视图直接发送到输出流，那么还可以使用 RenderPage 方法。RenderPage 方法使用视图的 URL 进行呈现。其总体作用与你在 ASPX 视图中可能大量使用过的 RenderPartial 扩展方法差不多。

4. 节的默认内容

Web Forms 视图引擎中的母版页可以让你指定一些用于占位符的默认内容，以防实际的页面未填写这些内容。这一特征本身不受 Razor 支持，但可以制定一些快速的解决方法。尤其是，WebViewPage 类提供了一个方便的 IsSectionDefined 方法，可以用在 Razor 模板中确定给定的节是否已被指定。这里有一些可以在布局页面中使用的代码，用以指明可选节的默认内容：

```
@* This code belongs to a layout page *@
<div id="footer">
    @if(IsSectionDefined("Copyright"))
    {
       @RenderSection("copyright")
    }
    else
    {
       <hr /><span>Rights reserved for a better use.</span>
    }
</div>
```

请记住节的名称不区分大小写。

5. 嵌套布局

可以对 Razor 布局进行任意程度的嵌套。假设你需要将由标准 ASP.NET MVC 应用程序所创建的 About 视图进行更为复杂的转换。About 视图基于一般的网站布局——_Layout.cshtml 文件。进一步假设你希望它接受来自外部视图模板的联系人信息。下面是你所期望的结构：

```
@{
    ViewBag.Title = "About Us";
}
<h2>About</h2>
<p>
     @RenderBody()
</p>
```

你不能直接请求布局模板——即任何名为 RenderBody 的 Razor 模板；一旦尝试直接请求，就会得到一个异常。这意味着你必须重命名 about.cshtml 文件，将其修改成类似于 aboutLayout.cshtml 的名称，如前面的代码中所示。另外，你必须显式地提及它所基于的父布局：

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "About Us";
}
<h2>About</h2>
<p>
    @RenderBody()
</p>
```

最后，可以创建实际的 about.cshtml 视图了，它是由应用程序直接请求的：

```
@{
    Layout = "~/Views/Home/AboutLayout.cshtml";
}
<fieldset>
    <legend>Contact</legend>
    <p>Follow me on Twitter: @@despos</p>
</fieldset>
```

6. 声明式 HTML 帮助器

如你在这一章前面所看到的，HTML 帮助器是构建可重用、参数化的 HTML 片段的有力工具。它是作为 HtmlHelper(或 AjaxHelper)类的扩展方法来编写的，可以轻松地在 Razor 模板中引用帮助器而不需要更改哪怕是一个代码。此规则适用于，比方说，所有内置的 HTML 帮助器，包括那些用于发出表单或操作链接的帮助器。

将 HTML 帮助器编写为扩展方法的缺点是，它们在表达图形布局方面的灵活性有限。例如，要生成一个 HTML 表格，需要将标记组合到一个字符串生成器中。可以这样做，但并不真正有效。在 Razor 中，你有了一个替代方案：声明式帮助器。

首先你要在该项目的 App_Code 文件夹中创建一个.cshtml 文件。项目向导不会自动创建该文件夹，所以你需要自己动手(奇怪的是，该文件夹甚至不会作为 ASP.NET 文件夹列示)。你为.cshtml 文件所选的名称很重要，因为在调用帮助器时会用到。可以将此文件夹看作是你的一个帮助器资源库。它的一个经典的名称类似于 MyHelpers.cshtml。下面是其中的一些内容：

```
@using RazorEngine.Models;

@helper CityGrid(IList<City> cities)
```

```
{
    <table>
        <thead> ... </thead>
        @foreach (var city in cities)
        {
            <tr>
                <td>@city.Name</td>
                <td>@city.Country</td>
                <td>@(city.Visited ? "Yes" : "No")</td>
            </tr>
        }
    </table>
}

@helper ShowHeader()
{
    <h2>I'm a Razor declarative helper!</h2>
}
```

@helper 关键字就是 HTML 声明的开头部分。紧跟关键字后面的是方法的签名和实现。帮助器主体只是 Razor 模板的一个嵌入式片段。可以在单个文件中有多个帮助器(请记住，如果帮助器资源库位于 App_Code 文件夹之外，那它不会被检测出来)。

要调用声明式 Razor 帮助器，请按照下面所显示的来做：

```
@using RazorEngine.Models;
@model IList<City>

@MyHelpers.ShowHeader()
<p>
    @MyHelpers.CityGrid(Model)
</p>
```

帮助器的名称由两部分组成：资源库名称和帮助器名称。

2.4　视图编码

在本章的最后一节，将深入探究影响视图和控制器的两个重要方面：如何有效地将数据传递给视图以及如何增加视图呈现处理的灵活性。

2.4.1　视图建模

使用 Razor(或 ASPX 引擎)，需要定义视图的图形布局。有时候，视图只由静态内容构成。

但更多时候，视图必须集成系统中间层操作所产生的实际数据，或者从应用程序缓存或会话状态中加载的数据。如何把这些数据提供给视图呢？

ASP.NET MVC 设计的黄金法则声明，视图会接收但不会计算它要显示的任何数据。可以通过三种非独占性的方式传递数据：ViewData 字典、ViewBag 字典以及量身定做的容器对象，也常被称作视图模型对象。

1. ViewData 字典

ViewData 属性直接由 Controller 类公开，它是名称-值字典对象。其编程模型类似于使用 Session 或其他 ASP.NET 内部对象：

```
public ActionResult Index()
{
    ViewData["PageTitle"] = "Programming ASP.NET MVC";
    return View();
}
```

你存储在字典中的任何数据都会被当作一个对象来处理，需要转换、装箱、或这两者一起以便被视图所用。可以按需在字典中创建条目。字典的生存周期与请求的生存周期是一样的。

ViewData 字典被封装到视图上下文中——一种内部结构，通过它 ASP.NET MVC 基础架构将数据从控制器级别提高到视图级别——并提供给视图引擎。视图对象——ASPX 引擎中的 ViewPage 和 Razor 中的 WebViewPage——向视图模板中的代码公开 ViewData 字典。以下显示了如何从视图模板检索 ViewData 内容：

```
<head>
  <title> @ViewData["PageTitle"] </title>
</head>
```

请记住你不会被限于仅在 ViewData 字典中存储字符串。

总体而言，ViewData 字典易于使用且非常灵活。实际上，它允许你通过直接创建一个新条目来把新的数据传递给视图。同时，基于名称的模型强制你使用大量的“魔幻字符串”(纯文本字符串，比如前面示例中的 PageTitle)，且更重要的是，要将这些字符串在控制器和视图代码之间匹配。通过使用常量，可以减少魔幻字符串的一些内在脆弱性，但你仍然无法避免抓取错误名字的可能。如果碰巧引用了错误的字典条目，则只有在运行时才能发现了。ViewData 字典非常适合于简单的解决方案和相对较短生命周期的应用程序。随着字典条目数量和视图数量的增长，维护会成为一个问题，因此寻找其他选项时应该摆脱 ViewData。

2. ViewBag 字典

ViewBag 属性也在 Controller 类上进行定义，它提供了一个更加灵活的将数据传递给视图的工具。该属性被定义为动态类型，如下所示：

```
public dynamic ViewBag { get; }
```

当.NET 编译器遇到一个动态类型时，它会发出特殊的代码块而不是简单地计算表达式。这种特殊的代码块会将表达式传递到动态语言运行时(Dynamic Language Runtime，DLR)用于运行时计算。换句话说，基于动态类型的任何表达式都会被编译成在运行时解释。从 ViewBag 中设置或读取的任何成员总是被编译器所接受，但其实是在执行时才会被计算。下面是一个比较 ViewData 和 ViewBag 用法的示例：

```
public ActionResult Index()
{
    // Using ViewData
    ViewData["PageTitle"] = "Programming ASP.NET MVC";

    // Using ViewBag
    ViewBag.PageTitle = "Programming ASP.NET MVC";

    return View();
}
```

编译器并不关心名为 PageTitle 的属性是否真的存在于 ViewBag 上。它所做的只是将调用打包给 DLR 解释器，在那里编译器要求 DLR 尝试为某特定 PageTitle 属性分配一个指定字符串。同样地，当从 ViewBag 中读取 PageTitle 时，编译器会指示 DLR 检查是否存在这样的属性。如果不存在，则编译器会抛出一个异常。以下代码显示了如何在 Razor 视图中使用 ViewBag 的内容：

```
<head>
   <title> @ViewBag.PageTitle </title>
</head>
```

从开发人员的角度来看，哪一个更好，ViewBag 还是 ViewData？

ViewBag 的语法比 ViewData 的语法更为简洁，但就我看来，这就是所有的区别了。和 ViewData 中的情形一样，不会在编译时检查属性。如果键入了错误的属性名称，那么由于动态类型对 DLR 的依赖，将不可避免地产生运行时异常。归根结底，这纯粹是由个人偏好所决定的。还有一个重要问题，ViewBag 至少需要是 ASP.NET MVC 3 和.NET 4，而 ViewData 的适用范围是 ASP.NET MVC 的任何版本和.NET 2.0。

注意：

动态类型是在运行时解析的，因此 Visual Studio 智能提示不能指示任何有关属性的内容。智能提示会把动态类型看作是普通的 Object 类型。有些工具——最引人注目的是 JetBrains ReSharper[②]——要更智能一些。ReSharper 会一路跟踪动态变量使用范围内所遇到的所有属性。对于任何使用过的属性，都会在智能提示菜单中添加一个条目。

3. 强类型视图模型

当你有几十个不同的值传递到视图时，快速添加新条目或重命名现有条目的灵活性就成为了最大的敌人。你只能自己跟踪项的名称和值；智能提示和编译器都帮不了你。

处理软件中复杂性的唯一经证明有效的方法是合适的设计。因此，为每个视图定义一个对象模型可以帮助你跟踪视图的真实所需。建议你为每一个添加到应用程序中的视图定义一个视图模型类：

```
public ActionResult Index()
{
    ...
    // Pack data for the view using a view-specific container object.
    var model = new YourViewModel();

    // Populate the model.
    ...

    // Trigger the view.
    return View(model);
}
```

每个视图都有一个视图模型类，这也会造成选择合适的类名称的问题。可以使用控制器名称和视图名称的一个组合。例如，从 Home 控制器调用的命名 Index 的视图，其视图模型对象可能被命名为 HomeIndexViewModel。更好的做法是，可以在 Models 文件夹中创建一个名为 Home 的子文件夹，并在其中托管一个 IndexViewModel 类。在我的应用程序中，我也常将 Models 重命名为 ViewModels(这种特殊方法只是一个建议；你应该自由地选择有意义的名称)。

那么如何开发一个视图模型类呢？

首先，视图模型对象是一个只有数据而(几乎)没有行为的普通数据传输对象。理想情况下，视图模型对象上的属性会完全以视图所期望的格式公开数据。例如，如果预计视图只显

② JetBrainsReSharper 是一款微软 Visual Studio 的插件，提供了智能 C#辅助编码功能和实时错误显示功能，并支持重构。

示待办订单的日期和状态，就不会想要传递一个完整 Order 对象的普通集合，因为它们是从中间层产生的。下面的视图模型类是视图用于数据建模的更好选择。它有助于保持表示层和中间层的解耦。

```
public class LatestOrderViewModel
{
    public DateTime OrderDate { get; set; }
    public String Status { get; set; }
}
```

ASP.NET MVC 基础架构保证了 ViewData 和 ViewBag 始终可用于视图对象而无需开发人员的任何干预。而自定义视图模型对象就不是这样了。

当你使用一个视图模型对象时，必须在视图模板中声明该视图模型类型，这样实际的视图对象就可以创建成 ASPX 视图引擎中的 ViewPage<T>类型；如果使用的是 Razor 的话，则会创建成 Razor 引擎中的 WebViewPage <T>类型。如果使用 ASPX Web Forms 视图引擎，下面就是需要在.aspx 模板中准备的内容：

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<LatestOrderViewModel>" %>
```

在前面的代码片段中，分配给 Inherits 特性的类型不是完全合格的。这意味着你可能需要添加一个@Import 指令来指定查找视图模型类型需导入的名称空间。如果视图引擎是 Razor，你所需要的是：

```
@model LatestOrderViewModel
```

要在视图模板中检索视图模型对象，可以使用在 WebViewPage 和 ViewPage 上都定义过的 Model 属性。下面是一个 Razor 的例子：

```
<h2>
    Latest order placed
    <span class="highlight">@Model.OrderDate.ToString("dddd, dd MMM yyyy")
                 </span>
    <br />
    Status is: <span class="highlight">@Model.Status</span>
</h2>
```

在该示例中，日期的格式是在视图中建立的。控制器使日期成为字符串，并将它传递给视图以备显示，这也是可以接受的。代码的归属问题并没有一个明确的标准；它掉进了一个灰色区域。我偏好尽量保持视图的简单。如果格式是固定的，不依赖运行时条件，那么传递 DateTime 而由视图解决其余的问题对你来说就是可行的。当设置日期格式需要有点逻辑的时候，我通常会将它移到控制器。

虽然每个视图都应该有自己的模型对象，但限制你要处理的类的数量通常是一个好选择。要在多个视图中重用模型类，你往往需要生成一个类的层次体系。下面是视图模型基类的一个示例：

```
public class ViewModelBase
{
    public String Title { get; set; }
    public String Header { get; set; }
}
public class LatestOrderViewModel : ViewModelBase
{
    ...
}
```

最后，围绕视图而非数据来设计视图模型类的结构往往是一个好办法。换句话说，我常常倾向于把视图模型类设计为一个容器。

假设你需要传递一个订单列表到视图。首先在头脑中出现的选项应该是使用下面的视图模型类(在 Razor 模板中)：

```
@model IList<PendingOrder>
```

从功能上讲，这个方法是可行的。但可扩展性呢？以我的经验来看，你最后总是需要在视图中填充各种异构数据。由于重构工作以及突然出现的新需求，这可能是必要的。下面是一个更易于维护的方法(允许你在不更改控制器/视图界面的情况下重构)：

```
@model YourViewModel
```

在本例中，YourViewModel 的定义如下所示：

```
public class YourViewModel
{
    public IList<PendingOrder> PendingOrders {get; set;}
}
```

视图模型类最终是为视图建模而不是为数据。

重要提示：

我不确定表达得是否足够清楚，所以换个方式表述一下吧。对于至少中等复杂程度和中等持续时间的任何 ASP.NET MVC 应用程序来说，强类型视图模型是唯一安全而可行的解决方法。我相信使用视图模型更多是一种心态而不是对抗复杂性的方式。但是，如果想方设法忽略了视图模型，那么为每个视图配备几个 ViewData 或 ViewBag 条目也能完成任务，但是几个月之后你就会弃用该站点(比如设置用于某一特定事件的站点)了。

4. 封装视图模型类

应该在何处定义视图模型类呢？这主要取决于项目的大小。在具有更好可重用性和预期更长使用寿命的大型项目中，你可能要以所使用的所有视图模型类来创建一个单独的类库。

在小项目中，你可能需要把所有的类隔离到一个特定的文件夹中。这可以是 Models 文件夹，这是为你所创建的默认 Visual Studio 项目模板。就个人而言，我倾向于将 Models 重命名为 ViewModels，并将其分组在控制器专属的子文件夹中，如图 2-6 所示。

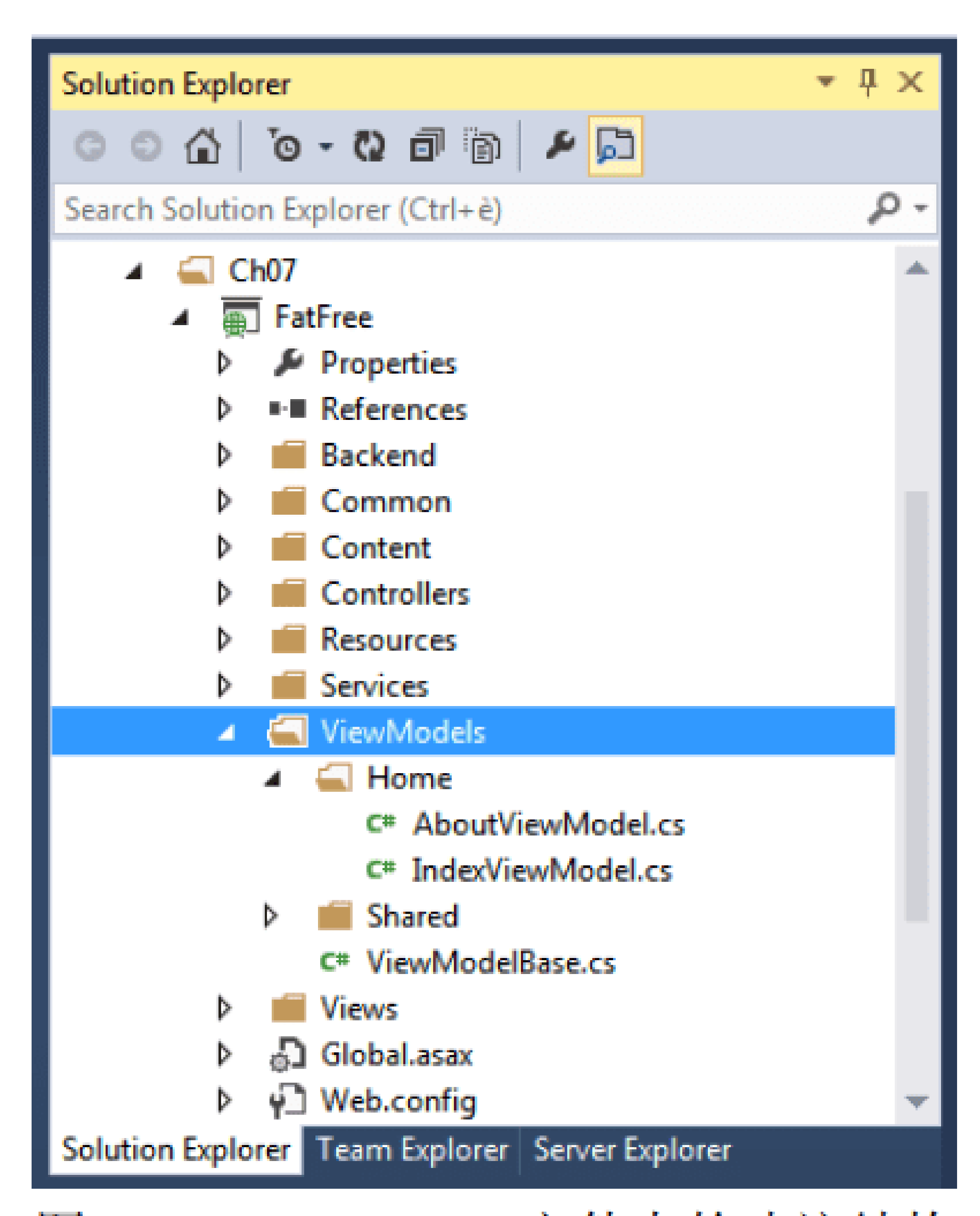


图 2-6　ViewModels 文件夹的建议结构

2.4.2　高级功能

ASP.NET MVC 的构建是基于约定优于配置的模式的。如同典型框架一样，这个模式为开发人员节省了很多的编程细节，只要他们遵守一些固定的规则和约定。这在视图中表现得尤其明显。

视图引擎的工作方式是检索和处理视图模板。它如何获知模板的情况呢？一般来说，它会以下面两种方式生效：用引擎注册已知的视图(配置)，或将视图放置在特定的位置，以便引擎可以检索它们(约定)。如果要依据一组不同的约定组织自己的视图呢？很简单，这需要你自己的视图引擎。

注意：

对自定义视图引擎的需求可能比最初想象的更为频繁。创建自定义视图引擎的主要原因大致有两个：希望以一种新的标记语言表示视图，或者你希望应用一套个性化约定。并不是说每个应用程序都该使用自定义的标记语言，但大多数应用程序都可能从以自定义方式组织的视图中受益。

1. 自定义视图引擎

我的大部分应用程序都采用它们自己的视图引擎，它们以略微不同的方式组织视图，或者需要一层额外的代码将视图名称解析成实际的标记文件。如果出于某些原因需要对一些视图使用不同的目录结构，那么只需要派生一个简单的类，如下所示：

```
public class MyViewEngine : RazorViewEngine
{
    public MyViewEngine()
    {
        this.MasterLocationFormats = base.MasterLocationFormats;
        this.ViewLocationFormats = new string[]
                                    {
                                      "~/Views/{1}/{0}.cshtml"
                                    };

        // Customize the location for partial views
        this.PartialViewLocationFormats = new string[]
                                    {
                                      "~/PartialViews/{1}/{0}.cshtml ",
                                      "~/PartialViews/{1}/{0}.vbhtml"
                                    };
    }
}
```

要用这个类代替默认的视图引擎，需要在 global.asax 中输入下面的内容：

```
protected void Application_Start()
{
    ...

    // Removes the default engines and adds the new one.
    ViewEngines.Engines.Clear();
    ViewEngines.Engines.Add(new MyViewEngine());
}
```

这样一来，如果任何一个部分视图不在 Partial Views 子文件夹中，你的应用程序都会出问题。

注意：

虽然就自定义视图引擎而言，设置自定义位置是有道理的，但你应该牢记如果要做的仅仅是设置其中一个默认引擎的位置格式属性，那真的不必创建一个自定义视图引擎。你可以直接在 Application_Start 中检索当前实例，以及设置位置格式属性。

重要提示：

如果有一个支持自定义文件夹(比如对部分视图进行分组的 PartialViews 文件夹)的自定义视图引擎，则向其添加一个 web.config 文件是十分必要的。可以复制在 Views 文件夹中所找到默认的与 web.config 相同的文件。该文件包含了使 ASP.NET MVC 运行时正确定位视图类的重要信息。

2. 呈现操作

复杂视图由子视图的各种组合产生而来。当控制器方法触发一个视图的呈现时，它必须提供视图的主结构和各相关部件所需的所有数据。有时候，这就要求控制器了解很多关于应用程序部件的详细情况，而这些信息可以使得类本身不用直接参与。我们来举例说明。

假设你有一个在很多视图中呈现的菜单。不论你进行任何与应用程序相关的操作，菜单都必须呈现。因此，呈现菜单是一个与当前正在处理的请求不直接相关的操作。你要如何应付此种情况呢？呈现操作是一个可行的答案。

呈现操作是专门设计用来从视图内部进行调用的控制器方法。因此呈现操作是控制器类上的一个常规方法，可以通过使用下面的一个 HTML 帮助器——Action 或 RenderAction——来从视图中进行调用。

```
@Html.Action("action")
```

Action 和 RenderAction 的作用差不多相同；唯一的区别是 Action 将标记作为字符串返回，而 RenderAction 直接写入输出流。这两种方法都支持多种重载，通过这些重载，可以指定多个参数，包括路由值、HTML 特性，当然还有控制器名称。可以将呈现操作定义为控制器类上的常规方法，也可以将其定义为某些视图相关操作的呈现程序：

```
public ActionResult Menu()
{
    var options = new MenuOptionsViewModel();
    options.Items.Add(new MenuOption {Url="...", Image="..."});
    options.Items.Add(new MenuOption {Url="...", Image="..."});
    return PartialView(options);
}
```

此处菜单的部分视图的内容是不相关的；它的作用只是获得模型对象和呈现适当的标记片段。我们看看应用程序中可能有的某个页面的视图源代码：

```
<div>
    ...
    @Html.RenderAction("Menu")
    ...
</div>
```

RenderAction 帮助器方法会调用指定控制器上(或命令呈现当前视图的控制器)的 Menu 方法，并引导对输出流的任何响应。用这种方式，视图会集成一些逻辑并回调该控制器。同时，你的控制器不需要担心视图信息的传递，它与当前处理的请求并不是严格相关的。

3. 子操作

呈现操作的执行并不是通过反射对一个方法进行调用那么简单。该过程中发生了很多事情。尤其是，呈现操作是在主要用户请求界限之内发起的子请求。RenderAction 方法会生成一个新的请求上下文，它包含与父请求相同的 HTTP 请求上下文和一组不同的路由值。这一子请求会被转发到一个特定的 HTTP 处理程序——ChildActionMvcHandler 类——并当作该子请求来自浏览器的情况进行处理。整体的运作类似于你调用服务器时的情况。会在普通的 ASP.NET 编程中执行。虽然没有重定向和往返，但子请求会通过常规 ASP.NET MVC 请求的通用管道，并接受操作筛选器的应用，这可能遭遇一些异常情况。其中，筛选器不会跨子操作工作的最常见的例子是 AuthorizeRequest 和 OutputCache(本书后面部分还会谈到操作筛选器)。

默认情况下，任何操作方法都可以从 URL 调用和通过呈现操作调用。但是任何以 ChildActionOnly 特性标记的操作方法都不会提供给公共调用方，对它们的使用仅限于呈现操作和子请求。

2.5　本章小结

ASP.NET MVC 不会将 URL 与磁盘文件相匹配；相反，它会解析 URL，找出要采取的下一个请求操作。每个操作都终止于一个操作结果。最常见的操作结果类型是视图结果，它是由大量 HTML 标记构成的。

视图结果由控制器方法生成，由模板和模型构成。视图引擎负责解析视图模板和填补模型数据。ASP.NET MVC 带有两个默认的视图引擎，它们各自支持不同的表示模板和在不同磁盘位置发现模板的标记语言。如今，Razor 是最常用的视图引擎，它取代了旧式的包含很多经典 ASP.NET 内容的 ASPX 语法，但它却不允许(这是合理的)使用服务器控件。

在这一章中，我们首先审视了处理视图的必要条件，然后着重于开发方面，包括使用 HTML 帮助器以及用于两个默认引擎 ASPX 和 Razor 的模板化帮助器。我们还讨论了对数据建模的最佳做法，并以强类型的视图模型对字典做了对比。

建议你看看同步学习源代码中的 BasicApp 示例，以便更好地理解本章所讨论的内容。在 ASP.NET MVC 5 中，由 Visual Studio 向导生成的默认应用程序是基于 Twitter(推特)引导程序的，并且提供了一个能适应不同屏幕尺寸的图形化模板。

第 3 章

模型绑定架构

不怕慢，就怕停。

——孔子

默认情况下，用于 ASP.NET MVC 应用程序的微软 Visual Studio 标准项目模板包含一个 Models 文件夹。如果查看一下它的使用指导以及其预期作用的有关信息，很快会得出 Models 文件夹是用于存储模型类的结论。那么，又是哪些模型呢？或者，更确切地说，“模型”的定义是什么？

我想说，“模型”是软件发展历史上被人误解最深的概念。它需要被扩展才能成为现代软件中的合理概念。在引入模型-视图-控制器(MVC)模式的时候，软件工程正处于起步阶段，应用程序也比今天要简单得多。没有人真正觉得有必要把模型的概念分解得更详细。然而这种更详细的情况却已经存在了。

我发现通常存在至少两种完全不同的模型：领域模型和视图模型。前者描述你在中间层使用的数据，预期会为填充业务领域的实体和关系提供可靠的表示。这些实体一般通过数据访问层来持久保存，并且通过实现业务流程的服务来使用。领域模型推动了数据的可视化，通常会使其更为鲜明，但同时可能与你在表示层发现的可见数据不同。视图模型只描述了表示层中正在处理的数据。一个典型例子就是标准的 Order 实体。可能有一些用例是应用程序需要向用户呈现订单集合，但并不是所有属性都要显示。比如，你可能需要显示 ID、日期、总额，以及一个独特的容器类——数据传输对象(DTO)。

话虽如此，但我同意并不是每个应用程序都必须将表示层与业务层中使用的对象模型严格分开。你可能出于自己的目的，决定让这两种模型基本重合，但应该始终意识到在两个不同层中运行着两种不同模型的事实。

这一章会介绍模型的第三种类型：输入模型，虽然它隐藏在 ASP.NET Web Forms 运行时之中多年，但在 ASP.NET MVC 中它被独立划分出来了。提交的数据会通过输入模型公开给控制器，随后被应用程序接收。输入模型定义了应用程序用来从输入表单处接收数据的 DTO。

注意：

还有一种类型的模型没有在这里提及，就是数据模型或者说用于持久保存数据的(主要是关系)模型。

3.1 输入模型

第 1 章“ASP.NET MVC 控制器”讨论了请求路由以及控制器方法的整体结构。第 2 章“ASP.NET MVC 视图”探讨了作为操作处理初步结果的视图。但是这两章都未能深入讨论在 ASP.NET MVC 中控制器方法是如何获取输入数据的。

在 ASP.NET Web Forms 中，我们有服务器控件、视图状态、还有在后台运行的整个页面生命周期来服务于准备好使用的输入数据。使用 ASP.NET Web Forms，开发人员无须担心输入模型。ASP.NET Web Forms 中的服务器控件为客户端用户界面提供了一个可靠的服务器端表示。开发人员只需要编写 C#代码从输入控件中读取即可。

ASP.NET MVC 强调用控制器接收而不是检索输入数据。要将输入数据传递给控制器，需要以某种方式把数据封装起来。而这正是输入模型发挥作用的时候。

为了更好地了解新的 ASP.NET MVC 输入模型的重要性和能力，我们首先从 ASP.NET Web Forms 的输入数据开始介绍。

3.1.1 Web Forms 输入处理的演变

ASP.NET Web Forms 应用程序是基于页面的，而每个服务器页面则基于服务器控件。页面具有自己的生命周期，从处理原始的请求数据时起，到组织用于浏览器的最终响应时止。页面的生命周期一路伴随着原始的请求数据，如 HTTP 标头、cookies、URL 和正文，并产生一个包含标头、cookies、内容类型和正文的原始 HTTP 响应。

在页面生命周期内部的几个步骤中，HTTP 原始数据会被传递进更易于编程的容器——服务器控件中。在 ASP.NET Web Forms 中，这些“可编程容器”永远不会作为输入对象模型的一部分来被识别。在 ASP.NET Web Forms 中，输入模型仅是基于服务器控件和视图状态的。

1. 服务器控件的作用

假设你有一个网页，它有几个 TextBox 控件来捕获用户名称和密码。当用户提交该表单的内容时，可能有一段类似于如下所示的代码来处理该请求：

```
public void Button1_Click(Object sender, EventArgs e)
{
  // You're about to perform requested action using input data.
  CheckUserCredentials(TextBox1.Text, TextBox2.Text);
```

```
    ...
}
```

ASP.NET Web Forms 架构的总体思路是使开发人员远离原始数据。任何传入的请求数据都被映射成服务器控件的属性。当无法使用这种方式时，数据会被留在通用容器里，如 QueryString 或 Form。

你对刚才所示的像 Button1_Click 这样的方法有什么期待？该方法是控制器操作的 Web Forms 对应方法。下面的内容介绍如何重构前面的代码以便能使用显式输入模型：

```
public void Button1_Click(Object sender, EventArgs e)
{
    // You're actually filling in the input model of the page.
    var model = new UserCredentialsInputModel();
    model.UserName = TextBox1.Text;
    model.Password = TextBox2.Text;

    // You're about to perform the requested action using input data.
    CheckUserCredentials(model);
    ...
}
```

ASP.NET 运行时环境会将原始的 HTTP 请求数据分解成控件属性，从而提供了以控件为中心的请求处理方法。

2. 视图状态的作用

从编程模式来说，ASP.NET Web Forms 和 ASP.NET MVC 的一个主要区别是视图状态。在 Web Forms 中，视图状态帮助服务器控件随时更新。正是因为有了视图状态，作为开发者，你不用去关注在回传过程中不会接触到的用户界面部分。假定你要显示一个可以下拉展开的选项列表。当提交详细信息的请求时，只需要在 Web Forms 中显示出这些详细信息即可。而原始的 HTTP 请求会提交这个选项列表以及所寻找的关键信息。视图状态使你不用亲自去处理这个选项列表。

视图状态和服务器控件在经典的 HTTP 机制之上构建了厚实的抽象层，它们让你从页面序列方面而不是连续的请求方面来考虑问题。这没什么对错可言；它只是 Web Forms 背后的模式而已。在 Web Forms 中，确实没有清楚定义输入模型的必要。如果这样做了，也只是因为你希望代码更为简洁和更具可读性。

3.1.2　ASP.NET MVC 中的输入处理

第 1 章阐述了控制器方法可以通过 Request 集合——如 QueryString、Headers 或 Form——或值提供程序来访问输入数据。这虽然很实用，但从可读性和维护的角度看这种方法并不理

想。你需要一个对控制器公开数据的专用模型。

1. 模型绑定器的作用

ASP.NET MVC 提供了一个自动绑定层，它使用一组内置的规则将原始的请求数据从任何一个值提供程序映射到输入模型类的属性。作为开发者，你要在很大程度上负责输入模型类的设计。就输入数据的处理而言，绑定层的逻辑可以在很大程度上进行自定义扩展，从而使灵活性达到前所未有的高度。

2. 模型的种类

ASP.NET MVC 的默认项目模板只提供了一个 Models 文件夹，因此可大致推论出“模型”的概念指的就是：应用程序应该使用的数据模型。大致上讲，这样的看法过于简单，但对于简单的网站来说是有效的。

如果看得更深入一些，就会认识到在 ASP.NET MVC 中有三种不同类型的“模型”，如图 3-1 所示。

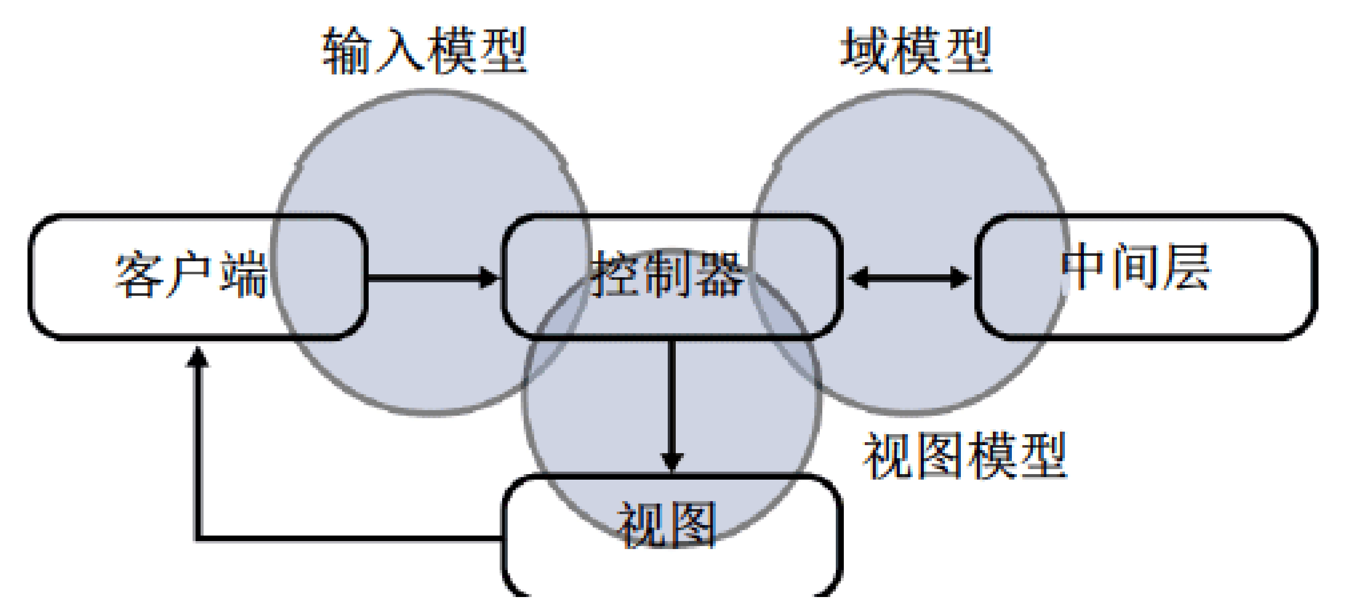


图 3-1　ASP.NET MVC 应用程序中可能涉及的模型类型

输入模型提供了正在提交到控制器的数据的表示。视图模型提供了正在视图中进行处理的数据的表示。最后，域模型是在中间层中操作的域特定实体的表示。

请注意这三种模型并不是完全分开的，图 3-1 在某种程度上显示出来了。你会发现模型之间有重叠。这意味着域模型中的类可能会用在视图中，从客户端提交的类也可能用在视图中。类的最终结构和关系图取决于你自己。

3.2　模型绑定

模型绑定是指将通过 HTTP 请求所提交的值绑定到控制器方法所用的参数的过程。我们来看看更多有关底层基础结构、机制和所涉及组件的内容。

3.2.1　模型绑定的基础结构

模型绑定逻辑是封装在一个特殊的模型绑定器类中的。绑定器在操作调用程序的控制下

工作，并帮助找出要传递给所选择的控制器方法的参数。

1. 分析方法的签名

第 1 章指出每一个传递到 ASP.NET MVC 的请求都会按照控制器名称和操作名称来解析。配备好这两块数据，操作调用程序——一个 ASP.NET MVC 运行时 shell 的原生组件——就可以发挥实际服务请求的作用了。首先，调用程序会把控制器名称扩展成一个类名称，并将操作名称解析为控制器类上的方法名称。如果发生错误，就会抛出异常。

然后，调用程序会收集进行方法调用所需的所有值。与此同时，它会查看方法的签名，试图找出签名中每个参数所需的输入值。

2. 为类型获取绑定器

操作调用程序知道每个参数的正式名称，并声明各个参数的类型(此信息是通过反射获得的)。操作调用程序还能够访问请求上下文以及与 HTTP 请求一起上载的任何数据——查询字符串、表单数据、路由参数、cookies、标头、文件等。

对于每一个参数，调用程序都会获得一个模型绑定器对象。模型绑定器是一个知道如何从请求上下文查找指定类型的值的组件。模型绑定器会应用它自己的算法，包括参数名称、参数类型和可用的请求上下文，并返回一个指定类型的值。算法的细节归属于该类型用到的模型绑定器的实现。

ASP.NET MVC 使用对应于 DefaultModelBinder 类的内置绑定器对象。模型绑定器是一个实现了 IModelBinder 接口的类。

```
public interface IModelBinder
{
    Object BindModel(ControllerContext controllerContext,
      ModelBindingContext bindingContext);
}
```

我们首先探讨默认绑定器的功能，然后看看为特定类型编写自定义绑定器需要些什么条件。

3.2.2　默认模型绑定器

默认模型绑定器使用基于规则的逻辑，将提交的值的名称与控制器方法中的参数名称相匹配。DefaultModelBinder 类知道如何处理简单和复杂类型，也清楚如何处理集合和字典。由此，默认绑定器大部分时间都能正常运行。

注意：

如果默认绑定器支持简单类型和复杂类型以及两者的集合，你觉得还有使用其他绑定器

的必要吗？你几乎不会认为有用另一种通用型绑定器来取代默认绑定器的需要。但是，默认绑定器不能应用你的自定义逻辑将请求数据传递到指定类型的属性中去。在后面的内容中你会看到，当请求所提交的值不能与你期望控制器所使用类型的属性完全相匹配时，自定义绑定器就非常有用了。在这种情况下，自定义的绑定器不仅有用，还能帮助保持控制器代码的至精至简。

1. 绑定简单类型

不可否认，似乎有些不可思议，但模型绑定背后的确没有什么神奇之处。模型绑定的关键一点在于，它使你可以将注意力完全放在你希望控制器方法所接收的数据上。可以完全忽略如何检索数据的细节，不论数据是来自于查询字符串还是路由。

假定你需要控制器方法按照给定的次数重复某个特定的字符串。下面就是你需要做的：

```
public class BindingController : Controller
{
  public ActionResult Repeat(String text, Int32 number)
  {
    var model = new RepeatViewModel {Number = number, Text = text};
    return View(model);
  }
}
```

像这样设计，控制器就具有了高度可检验性，并能从 ASP.NET 运行时环境完全解耦。因此你不用直接访问 Request 对象或 Cookies 集合。

文本和数字的值是从哪儿来的呢？具体又是哪个组件将它们读入文本和数字参数的呢？

从请求上下文读取的实际值和默认模型绑定器对象是起作用的关键。尤其是，默认绑定器会试图将形参的名称(本例中是文本和数字)与随请求提交的名称值相匹配。也就是说，如果请求携带有表单字段、查询字符串字段或者名称为文本的路由参数，那么携带的值就会自动绑定到文本参数。只要参数类型和实际值是兼容的，映射就会成功。如果不能执行转换，则会抛出参数异常。下面的 URL 可以正常使用：

```
http://server/binding/repeat?text=Dino&number=2
```

相反，下面的 URL 会引发异常：

```
http://server/binding/repeat?text=Dino&number=true
```

上述查询字符串字段文本包含 Dino，其到 Repeat 方法上的 String 文本参数的映射是成功的。另一方面，查询字符串字段的数字包含 true，因此不可能成功映射到一个 Int32 参数。模

型绑定器返回一个其中的 number 条目包含 null 的参数字典。因为该参数的类型是 Int32——也就是非空的类型——因此调用程序会抛出参数异常。

2. 处理可选值

由于传递无效值而抛出的参数异常不会在控制器级别检测出来。异常在执行流程到达控制器之前就触发了。这意味着你不可能用 try/catch 块来捕获它。

如果默认模型绑定器无法找到与所需方法参数相匹配的提交值，那么它会在参数字典中放置一个 null 值，该参数字典会返回给操作调用程序。同样的，如果 null 值不能为参数类型所接受，那么在控制器方法被调用之前就会被抛出一个参数异常。

如果必须把方法参数作为可选项呢？

一个可行的办法是将参数的类型更改为允许空值的类型，如下所示：

```
public ActionResult Repeat(String text, Nullable<Int32> number)
{
    var model = new RepeatViewModel {Number = number.GetValueOrDefault(),
                   Text = text};
    return View(model);
}
```

另一种方法包含了参数的默认值使用：

```
public ActionResult Repeat(String text, Int32 number=4)
{
    var model = new RepeatViewModel {Number = number, Text = text};
    return View(model);
}
```

控制器方法的签名由你自己决定。一般情况下，你可能希望使用接近于随请求上传的真实数据的类型。例如，使用 Object 参数类型就不会引发参数异常，但这将导致难以编写简洁的代码来处理输入的数据。

默认绑定器可以映射所有简单类型，如 String、整数类型、Double、Decimal、Boolean、DateTime 以及相关的集合。要在 URL 中表示 Boolean 类型，需要借助于 true 或 false 字符串。这些字符串使用.NET 原生 Boolean 解析函数解析，能够不区分大小写地识别 true 和 false 字符串。如果使用 yes/no 这样的字符串表示 Boolean，默认绑定器是不能识别你的意图的，它会在参数字典中发出一个 null 值，从而导致参数异常。

3. 值提供程序和优先级

默认模型绑定器会使用所有已注册的值提供程序将提交值与方法参数进行匹配。默认情况下，值提供程序涵盖了表 3-1 中所列出的集合。

表 3-1　默认值提供程序所涵盖的请求集合

集　　合	描　　述
Form	如果有的话，则会包含从 HTML 表单提交的值
RouteData	包含摘自 URL 路由的值
QueryString	包含指定为 URL 查询字符串的值
Files	如果有的话，则值为上传文件的所有内容

表 3-1 列出了需要在默认绑定器中按照确切顺序进行处理的请求集合。假设你有下面的路由：

```
routes.MapRoute(
    "Test",
    "{controller}/{action}/test/{number}",
    new { controller = "Binding", action = "RepeatWithPrecedence",
          number = 5 }
);
```

可以看到，该路由具有一个名为 number 的参数。现在思考下面这个 URL：

```
/Binding/RepeatWithPrecedence/test/10?text=Dino&number=2
```

该请求会上传两个值，它们是在 RepeatWithPrecedence 方法中设置 number 参数的值的理想对象。第一个值是 10，是名为 number 的路由参数的值。第二个值是 2，是名为 number 的 QueryString 元素的值。该方法自身提供了一个数字参数的默认值：

```
public ActionResult RepeatWithPrecedence(String text, Int32 number=20)
{
  ...
}
```

实际上选中了哪个值呢？表 3-1 表明，实际上传递给该方法的值是 10，它是从路由数据集合中读取的值。

4. 绑定复杂类型

可以在方法签名上列出的参数的数目没有上限。但是，一个容器类往往比一个各种参数的长列表要好。对于默认模型绑定器，无论你列出一系列参数，还是复杂类型的一个参数，结果几乎相同。这两种方案都完全受到支持。下面是一个示例：

```
public class ComplexController : Controller
{
   public ActionResult Repeat(RepeatText inputModel)
```

```
        {
           var model = new RepeatViewModel
                           {
                             Title = "Repeating text",
                             Text = inputModel.Text,
                             Number = inputModel.Number
                           };
          return View(model);
        }
    }
```

该控制器方法会接收一个 RepeatText 类型的对象。该类是一个普通的数据传输对象，其定义如下：

```
public class RepeatText
{
  public String Text { get; set; }
  public Int32 Number { get; set; }
}
```

可以看到，这个类只包含与前面示例中作为单个参数传递的值相同的值。模型绑定器能够处理该复杂类型，就像它处理单个值的情况一样。

对于声明的类型(本例中是 RepeatText)中的每一个公共属性，模型绑定器都会寻找其关键字名称与属性名称相匹配的提交值。匹配不区分大小写。下面是一个使用 RepeatText 参数类型的示例 URL：

```
http://server/Complex/Repeat?text=ASP.NET%20MVC&number=5
```

图 3-2 显示了该 URL 可能会生成的输出。

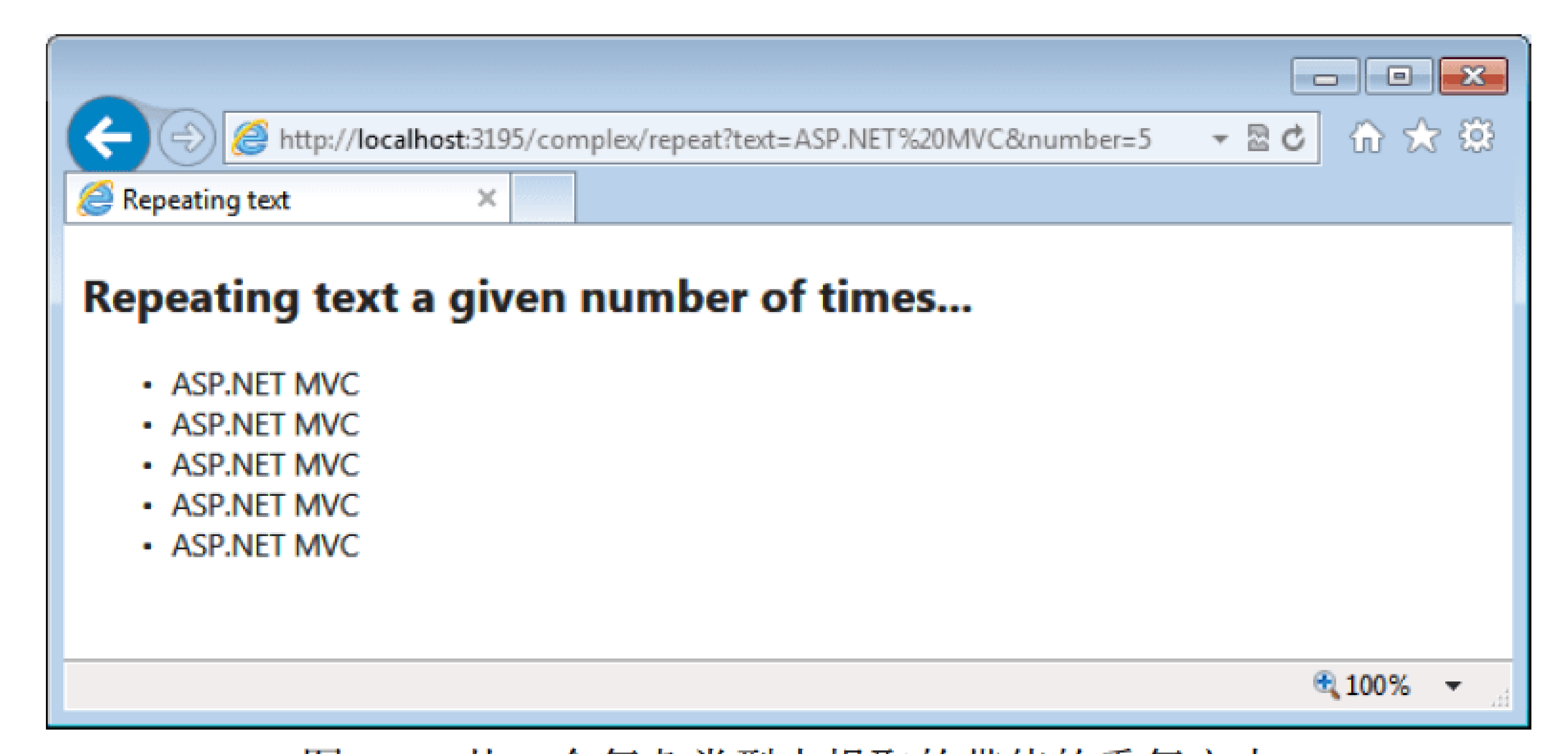


图 3-2　从一个复杂类型中提取的带值的重复文本

5. 绑定集合

如果控制器方法预期的参数是一个集合呢？比如，可以把提交表单的内容绑定到一个 IList <T>参数吗？DefaultModelBinder 类可以帮助你做到，但这样做需要你自己想点办法。看看图 3-3。

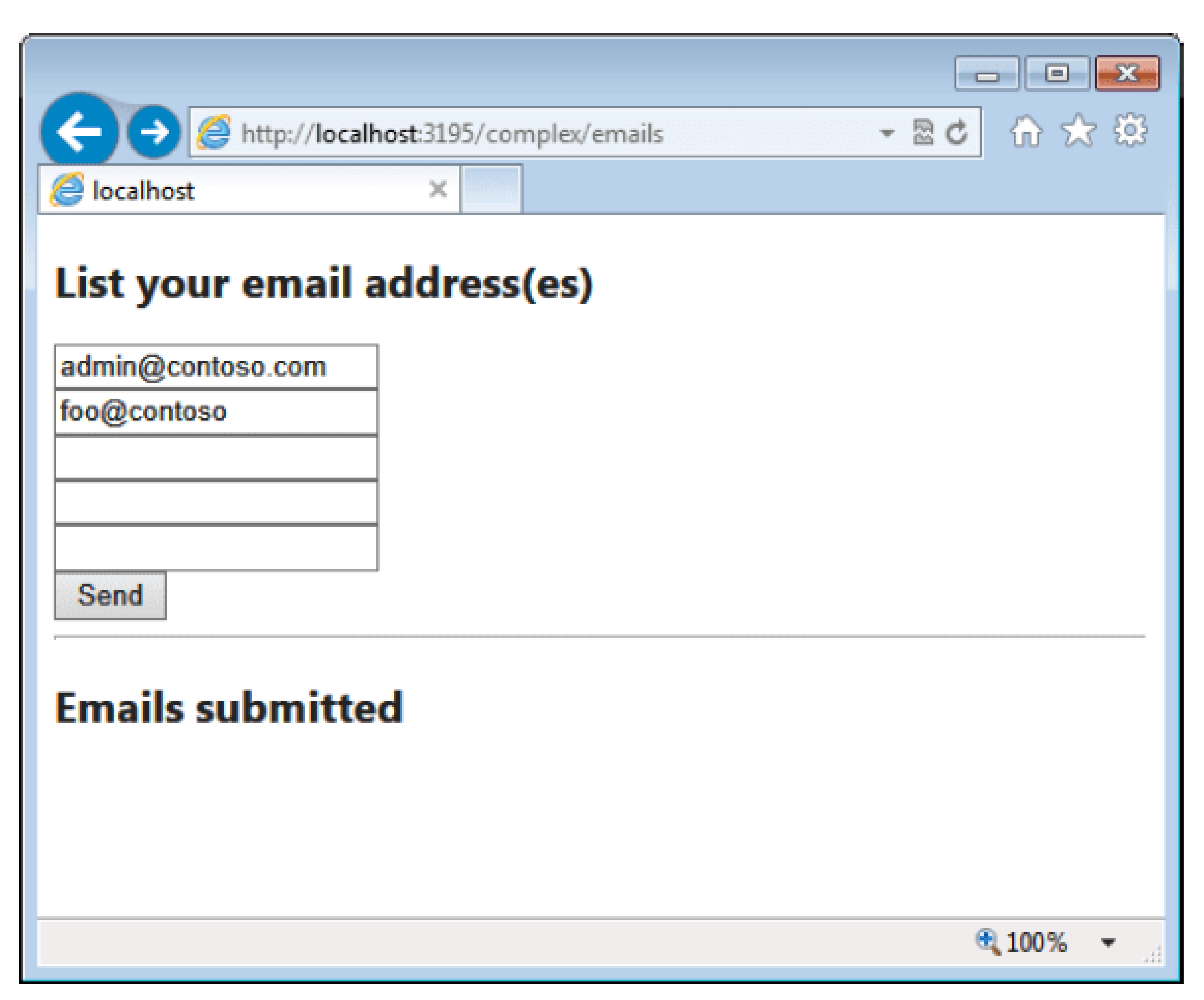


图 3-3　该页面会提交一个字符串数组

当用户单击 Send 按钮时，该表单会提交内容。具体来说，它会发送不同文本框的内容。如果这些文本框具有不同的 ID，那么提交的内容会采用以下形式：

```
TextBox1=admin@contoso.com&TextBox2=&TextBox3=&TextBox4=&TextBox5=
```

在经典的 ASP.NET 中，这是唯一行得通的办法，因为你不能把同一个 ID 分配给多个控件。但是，如果自己管理 HTML 的话，是可以将该图中五个文本框分配到同一 ID 的。事实上，HTML DOM 完全支持这种情况(虽然不被推荐)。因此，下面的标记在 ASP.NET MVC 中是完全合法的，并且会生成对所有浏览器都有效的 HTML：

```
@using (Html.BeginForm())
{
    <h2>List your email address(es)</h2>
    foreach(var email in Model.Emails)
    {
        <input type="text" name="email" value="@email" />
        <br />
    }
    <input type="submit" value="Send" />
}
```

必须处理在表单中所键入的 email 地址的控制器方法的预计签名是什么？是下面这个：

```
public ActionResult Emails(IList<String> email)
{
  ...
}
```

图 3-4 显示出，正是由于有了默认绑定器类，一个字符串数组就可以正确地传递给该方法。

图 3-4　一个已提交的字符串数组

正如将在第 4 章“输入表单”中详尽讨论的，当你使用 HTML 表单时，你很可能需要具有两个方法：一个处理视图的显示(GET 动词)，另一个处理将数据发送到视图的情况。HttpPost 和 HttpGet 特性使你能够将指定方法处理相同操作名称的情形标记出来。下面是该示例的完整实现，它使用了两种不同的方法处理 GET 和 POST 的情况：

```
 [ActionName("Emails")]
[HttpGet]
public ActionResult EmailForGet(IList<String> emails)
{
    // Input parameters
    var defaultEmails = new[] { "admin@contoso.com", "", "", "", "" };
    if (emails == null)
        emails = defaultEmails;
    if (emails.Count == 0)
        emails = defaultEmails;
    var model = new EmailsViewModel {Emails = emails};
    return View(model);
}

[ActionName("Emails")]
[HttpPost]
public ActionResult EmailForPost(IList<String> email)
{
```

```
    var defaultEmails = new[] { "admin@contoso.com", "", "", "", "" };
    var model = new EmailsViewModel { Emails = defaultEmails, RegisteredEmails
             = email };
    return View(model);
}
```

下面是图 3-5 中所呈现的视图的完整 Razor 标记：

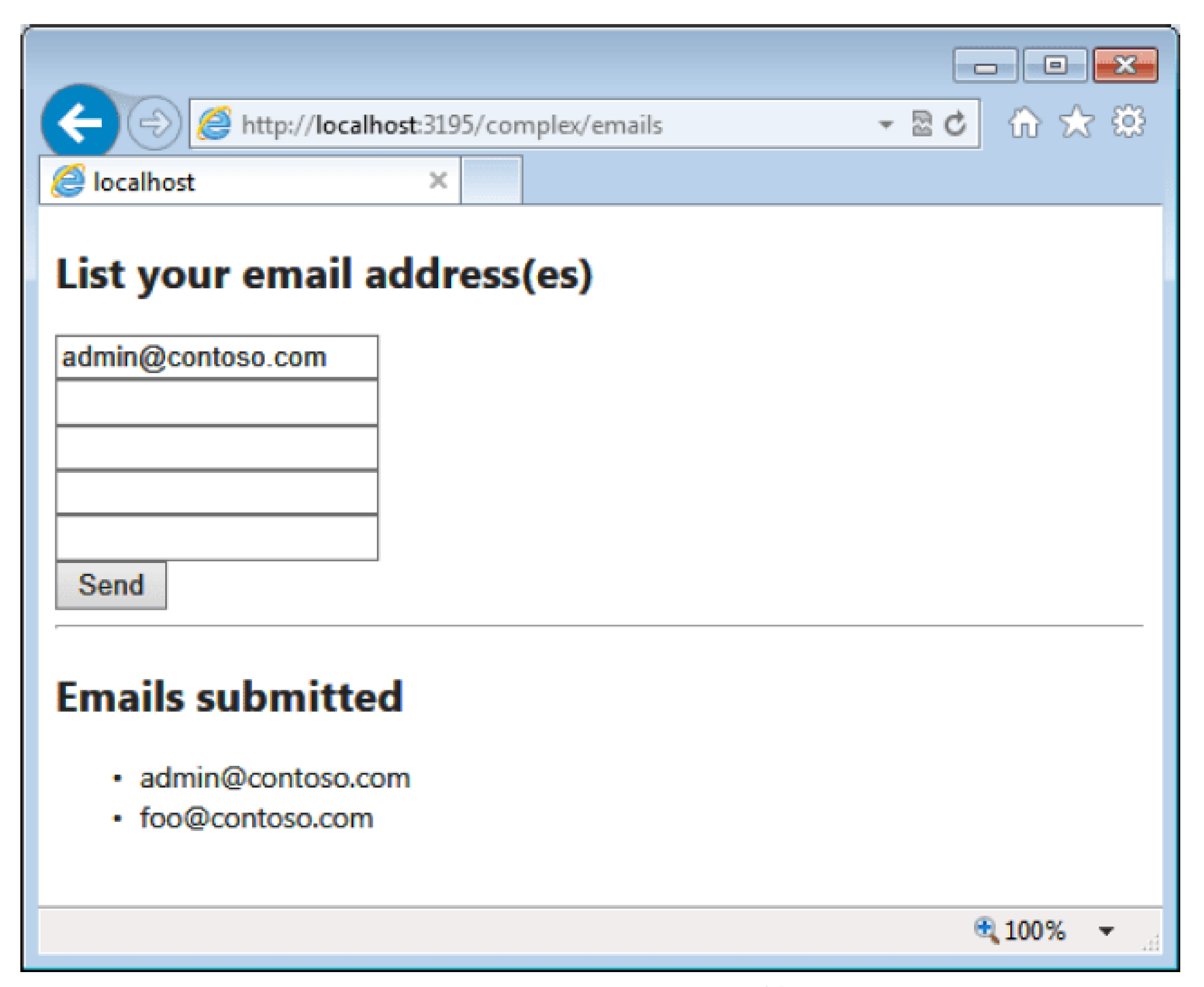


图 3-5　POST 之后呈现的页面

```
@model BindingFun.ViewModels.Complex.EmailsViewModel

<h2>List your email address(es)</h2>
@using (Html.BeginForm())
{
    foreach(var email in Model.Emails)
    {
        <input type="text" name="email" value="@email" />
        <br />
    }
    <input type="submit" value="Send" />
}

<hr />
<h2>Emails submitted</h2>
<ul>
@foreach (var email in Model.RegisteredEmails)
{
```

```
        if (String.IsNullOrWhiteSpace(email))
        {
            continue;
        }
        <li>@email</li>
    }
    </ul>
```

最后，为了保证值的集合能够传递到控制器方法，你需要确保具有相同 ID 的元素都被发送到响应流。然后，该 ID 必须依据绑定器的常规规则匹配到控制器的方法签名。

如果是集合的情况，名字之间所需的匹配会使你不得不违背基本的命名规则。在视图中，你拥有输入字段并希望调用它们，比如，email 使用的是单数形式。当你命名控制器中的参数时，由于你要得到的是集合，因此你可能会将其命名为 emails 这一复数形式。然而，你不得不至始至终要么使用 email 要么使用 emails。变通办法将在稍后我们介绍到考虑模型绑定器的自定义方面时谈到。

6. 绑定复杂类型的集合

默认绑定器也可以处理集合中包含复杂类型的情况，甚至是如下所示的嵌套情形：

```
 [ActionName("Countries")]
[HttpPost]
public ActionResult CountriesForPost(IList<Country> country)
{
   ...
}
```

举例来说，请思考下面 Country 类型的定义：

```
public class Country
{
    public Country()
    {
        Details = new CountryInfo();
    }
    public String Name { get; set; }
    public CountryInfo Details { get; set; }
}
public class CountryInfo
{
    public String Capital { get; set; }
    public String Continent { get; set; }
}
```

要使模型绑定成功地进行，你真正需要做的就是对标记中的 ID 使用渐进式索引。生成的模式是 prefix[index]。Prefix 所代表的属性会匹配控制器方法的签名中的形参名称：

```
@using (Html.BeginForm())
{
    <h2>Select your favorite countries</h2>
    var index = 0;
    foreach (var country in Model.CountryList)
    {
        <fieldset>
        <div>
            <b>Name</b><br />
            <input type="text"
                   name="countries[@index].Name"
                   value="@country.Name" /><br />
            <b>Capital</b><br />
            <input type="text"
                   name="country[@index].Details.Capital"
                   value="@country.Details.Capital" /><br />
            <b>Continent</b><br />
            @{
                var id = String.Format("country[{0}].Details.Continent",
index++);
             }
             @Html.TextBox(id, country.Details.Continent)
             <br />
        </div>
    </fieldset>
    }
    <input type="submit" value="Send" />
}
```

索引是数值型，从 0 开始渐进的。在这个示例中，为每个指定的默认国家构建用户界面块。如果有固定数量的用户界面块要呈现，可以使用静态索引。

```
<input type="text"
       name="country[0].Name"
       value="@country.Name" />

<input type="text"
       name="country[1].Name"
       value="@country.Name" />
```

注意系列中的间隙(比如 0 与 2 之间)会停止解析过程，你重新获得的是从 0 到这个间隙

的数据类型序列。

数据的提交也会正常执行。控制器类上的 POST 方法只会接收相同层级的数据，如图 3-6 所示。

```
[HttpPost]
[ActionName("Countries")]
public ActionResult CountriesForPost(IList<Country> country)
{
    var defaultCountries = GetDefaultCountries();

    var model = new CountriesViewModel
    {
        CountryList = defaultCountries,
        SelectedCountries = country
    };
    return View(model);
}
```

country　Count = 3
[0]　{BindingFun.InputModels.Country}
[1]　{BindingFun.InputModels.Country}
Details　{BindingFun.InputModels.CountryInfo}
Name　"Spain"

图 3-6　提交到该方法的复杂类型和嵌套类型

如果提交的值在映射到预期类型层级的过程中遇到麻烦，不用担心，可以考虑使用自定义的模型绑定器。

7. 绑定上传文件的内容

表 3-1 表明上传的文件也可以成为模型绑定的主体。默认绑定器通过匹配与参数名称一起上传的输入文件元素的名称来实现绑定。但是参数(或参数类型上的属性)必须声明为 HttpPostedFileBase 类型：

```
public class UserData
{
    public String Name { get; set; }
    public String Email { get; set; }
    public HttpPostedFileBase Picture { get; set; }
}
```

下面的代码显示了将上传文件保存在服务器计算机中某个位置的控制器操作的一种可能实现：

```
public ActionResult Add(UserData inputModel)
{
    var destinationFolder = Server.MapPath("/Users");
    var postedFile = inputModel.Picture;
    if (postedFile.ContentLength > 0)
    {
        var fileName = Path.GetFileName(postedFile.FileName);
        var path = Path.Combine(destinationFolder, fileName);
        postedFile.SaveAs(path);
    }
```

```
        return View();
    }
```

默认情况下，任何 ASP.NET 请求都不能超过 4MB。这个数字包含了所有的上传、标头、正文以及正在传输的任何内容。可以通过 web.config 文件中 httpRuntime 部分的 maxRequestLength 条目来配置各个级别的上传阈值：

```
<system.web>
  <httpRuntime maxRequestLength="6000" />
</system.web>
```

很明显，请求越大，可能你留给黑客攻击网站的空间也越大。同时请记住在托管方案中，如果主机设置了域级别的不同限制，并锁定了较低级别的 maxRequestLength 属性，那么你的应用程序级别的设置可能会被忽略。

那么上传多个文件的情形呢？只要上传的整体规模与当前的最大请求长度相符，你是可以在单个请求中上传多个文件的。然而，需要考虑的是 Web 浏览器并不知道如何上传多个文件。Web 浏览器能做的只是上传一个文件，并且只能通过一个 file 类型的输入元素引用它才行。要上传多个文件，可以借助于一些客户端专用组件，或在表单中放置多个<input>元素。如果使用多个正确命名的<input>元素，一个如下所示的类会将它们全部绑定：

```
public class UserData
{
    public String Name { get; set; }
    public String Email { get; set; }
    public HttpPostedFileBase Picture { get; set; }
    public IList<HttpPostedFileBase> AlternatePictures { get; set; }
}
```

这个类表示了提交的用于新用户的数据，其中带有一个默认图片和一系列备用图片。下面是备用图片的标记：

```
<input type="file" id="AlternatePictures[0]" name="AlternatePictures[0]" />
<input type="file" id="AlternatePictures[1]" name="AlternatePictures[1]" />
```

ASP.NET 应用程序账户

在 Web 服务器上创建文件通常不是依赖默认权限集就可以完成的操作。所有 ASP.NET 应用程序都是在那些为应用程序所属的应用程序池提供服务的工作进程账户下运行的。在正常情况下，这个账户是 NETWORK SERVICE，而它并未获得创建新文件的权限。这就是说为了保存文件，你必须更改 ASP.NET 应用程序使用的账户，或者提升默认账户的权限。

> 多年来，应用程序池的标识一直是一个固定的标识——也即前面所说的 NETWORK SERVICE 账户，它在微软 Windows 中是一个权限相对较低的内置标识。它最初作为出色的安全措施是受欢迎的，但是到后来，使用单个账户来应对潜在的众多并发运行服务的做法引发了很多问题，与它所解决的问题相比就有些得不偿失了。
>
> 总而言之，在相同账户下运行的服务可能会相互干涉。为此，在 Internet Information Services 7.5 中，工作进程默认会在唯一标识下运行，这些唯一标识是自动为每个新创建的应用程序池创建的。其底层技术是虚拟账户(Virtual Accounts)，目前受到 Windows Server 2008 R2 和 Windows 7 以及更高版本的支持。更多有关信息，请查看 http://technet.microsoft.com/library/dd548356.aspx。

3.2.3　默认绑定器的可自定义方面

自动绑定源于约定优于配置的方式。但有时候这些约定包含不好的意外情况。如果出于某些原因你失去了对所提交数据的控制(比如，数据被篡改的情况)，可能会导致不希望发生的绑定；事实上，任何发送的键/值对都会被绑定。出于此原因，你应该考虑使用 Bind 特性对绑定过程的某些方面进行自定义。

1. Bind 特性

Bind 特性包含三个属性，如表 3-2 中所示。

表 3-2　BindAttribute 类的属性

属　　性	描　　述
Prefix	字符串属性。它表示用于绑定器解析的必须在提交值名称中发现的前缀。其默认值是空字符串
Exclude	获得或设置不允许绑定的用逗号分隔的属性名称列表
Include	获得或设置允许绑定的用逗号分隔的属性名称列表

需要将 Bind 特性应用在方法签名的参数上。

2. 创建属性白名单

前面说过，当你有复杂类型的时候，自动模型绑定是有潜在风险的。在这种情况下，只要默认绑定器在提交值中找到了匹配项，它就会试图把所有的公共属性填入复杂类型中。这可能最终导致服务器类型充满了意料之外的数据，尤其是在请求被篡改的情况下。为了避免此种情况的出现，可以使用 Bind 特性上的 Include 属性，创建一个可接受的属性白名单，如

下所示：

```
public ActionResult RepeatOnlyText([Bind(Include = "text")]RepeatText
        inputModel)
{
  ...
}
```

RepeatText 类型的绑定将限于所列出的属性(对于上面的示例，只有 Text 属性被允许)。其他的所有属性均不会绑定，而是采用在 RepeatText 实现中分配给它们的默认值。多个属性之间以逗号分隔。

3. 创建属性黑名单

Exclude 特性采用相反的逻辑：它罗列出了必须从绑定中排除的属性。除了那些显式列出的属性，其他属性都会被绑定：

```
public ActionResult RepeatOnlyText([Bind(Exclude = "number")]RepeatText
        inputModel)
{
  ...
}
```

可以在相同的特性中同时使用 Include 和 Exclude，如果这样能够使你更好地定义要绑定的属性集的话。例如，如果两个特性引用相同的属性，那么 Exclude 会生效。

4. 加前缀的参数别名

默认的模型绑定器会强制你将请求参数(比如，表单和查询字符串字段)命名为在目标操作方法上与形参相匹配的指定名称。使用 Prefix 特性，可以更改此规则。通过设置 Prefix 特性指示模型绑定器：请求参数要匹配形参的前缀，而不是形参名称。总之，别名对于特性来说应该是一个更好的名称。请思考下面的示例：

```
[HttpPost]
[ActionName("Emails")]
public ActionResult EmailForPost([Bind(Prefix = "email")]IList<String>
        emails)
{
   ...
}
```

为了成功填充 emails 参数，你需要提交名称是 email 而非 emails 的字段。Prefix 特性对于 POST 方法具有特殊意义，它解决了上述命名规则和参数集合的问题。

最后，需要注意的是如果指定了前缀，那么它将成为强制性的；随后，未添加前缀的名称字段都不会被绑定。

注意：

为特性 Prefix 选择的名称确实并非它所应对情形的真实说明。大家都认为别名本来可以是一个更好的名称。但是，现在想改变太迟了！

3.3　高级模型绑定

到目前为止，我们探究了默认模型绑定器的作用。默认绑定器发挥了出色的作用，但它是一个通用工具，旨在通过某种通用方式处理最常见的类型。Bind 特性使你对绑定过程有了更多的控制，但它的能力也受到了一些合理的限制。如果想要实现对绑定过程的完全控制权，则需要创建针对不同类型的自定义绑定器。

3.3.1　自定义类型绑定器

只有一个重要原因使你愿意创建自定义的绑定器：即默认绑定器仅限于处理提交值与模型属性之间一对一的对应关系。

然而有时候，目标模型有着与表单字段所表示的不同的粒度。典型的例子是，当你使用多个输入字段来让用户输入单个属性内容的时候；例如，不同的输入字段为日、月和年，然后映射到一个单一的 DateTime 值。

1. 自定义默认绑定器

要从头开始创建一个自定义的绑定器，你需要实现 IModelBinder 接口。想要实现对绑定过程的完全控制，推荐实现这个接口。比如，如果只需要保持默认的功能，并且仅仅强制绑定器使用一个用于指定类型的非默认构造函数，那么从 DefaultModelBinder 继承是最好的办法。下面是要遵循的架构：

```
public RepeatTextModelBinder : DefaultModelBinder
{
    protected override object CreateModel(
        ControllerContext controllerContext,
        ModelBindingContext bindingContext,
        Type modelType)
    {
        ...
        return new RepeatText( ... );
    }
}
```

单纯重写默认绑定器的另一个常见情形是，当你所需的只是针对特定类型进行验证的能力时。在这种情况下，你只需要重写 OnModelUpdated 并插入你自己的验证逻辑，如下所示：

```
protected override void OnModelUpdated(ControllerContext controllerContext,
         ModelBindingContext bindingContext)
{
   var obj = bindingContext.Model as RepeatText;
   if (obj == null)
     return;

   // Apply validation logic here for the whole model
   if (String.IsNullOrEmpty(obj.Text))
   {
     bindingContext.ModelState.AddModelError("Text", ...);
   }
   ...
}
```

如果希望各个属性的所有验证都保持在一个位置，则可以重写 OnModelUpdated。如果更倾向于单独放置验证属性，则可以借助于 OnPropertyValidating。

重要提示：

当绑定发生于复杂类型时，默认绑定器会直接将匹配的值复制到属性中。如果其中的某些值使复杂类型的实例处于无效状态的话，你无法拒绝匹配它们。

自定义绑定器可以集成一些逻辑来检查分配给属性的值，并且向控制器方法表明产生了错误，或者通过用默认值替换无效值从而大大减轻错误程度。

虽然这种方法可以使用，但它并没有被普遍采用，这是因为在 ASP.NET MVC 中有更强大的选项可用来处理输入表单中所有的数据验证。这也是我将在第 4 章讨论的话题。

2. 从头开始实现模型绑定器

IModelBinder 接口的定义如下：

```
public interface IModelBinder
{
  Object BindModel(ControllerContext controllerContext,
                   ModelBindingContext bindingContext);
}
```

下面是直接实现 IModelBinder 接口的自定义绑定器的框架。该模型绑定器是为一个特定的类型编写的——本例中为 MyComplexType：

```
public class MyComplexTypeModelBinder : IModelBinder
```

```
{
  public Object BindModel(ControllerContext controllerContext,
                          ModelBindingContext bindingContext)
  {
      if (bindingContext == null)
         throw new ArgumentNullException("bindingContext");

      // Create the model instance (using the ctor you like best)
      var obj = new MyComplexType();

      // Set properties reading values from registered value providers
      obj.SomeProperty = FromPostedData<string>(bindingContext,
"SomeProperty");
      ...
      return obj;
  }
  // Helper routine
  private T FromPostedData<T>(ModelBindingContext context, String key)
  {
     // Get the value from any of the input collections
     ValueProviderResult result;
     context.ValueProvider.TryGetValue(key, out result);

     // Set the state of the model property resulting from value
     context.ModelState.SetModelValue(key, result);

     // Return the value converted (if possible) to the target type
     return (T) result.ConvertTo(typeof(T));
  }
```

BindModel 类的结构非常简单。首先创建一个要用到的 BindModel 类的新实例。为此，可以使用喜欢的构造函数(或工厂)并且执行上下文所需的自定义初始化。接着，只需要为刚创建的实例的属性填入从提交数据中读取或推测的值。在前面的代码片段中，假设仅仅复制了默认提供程序的行为，并且从基于匹配属性名称的已注册值提供程序中读取值。很明显，这就是你需要添加自己的逻辑的地方，来解释和传递由请求上传的内容。

请记住，编写模型绑定器时，你并不局限于仅从提交数据中获取用于模型的信息，虽然这代表了大多数常见情形。可以从任何位置(比如从 ASP.NET 缓存和会话状态中)获取信息，解析并将它存储在模型中。

注意：

除了默认绑定器，ASP.NET MVC 还带有两个内置的绑定器。这两个额外的绑定器分别在提交数据是 Base64 流(ByteArrayModelBinder 类型)以及文件内容正在上传

(HttpPostedFileBaseModelBinder 类型)时自动选用。

3. 注册自定义绑定器

可以将模型绑定器与其目标类型进行全局关联或局部关联。对于全局关联，所有用于某类型的模型绑定器的匹配都会通过已注册的自定义绑定器加以解析。而对于局部关联，绑定会应用于控制器方法中某个参数的一次匹配。

全局关联在 global.asax 文件中进行定义，如下所示：

```
void Application_Start()
{
  ...
  ModelBinders.Binders[typeof(MyComplexTypeModelBinder)] =
                          new MyCustomTypeModelBinder();
}
```

局部关联要用下面的语法：

```
public ActionResult RepeatText(
        [ModelBinder(typeof(MyComplexTypeModelBinder))] MyComplexType info)
{
  ...
}
```

局部绑定器总是优先于全局定义的绑定器。

可以从前面代码的 Application_Start 中清晰地获知，可以注册多个绑定器。如果需要，还可以重写默认的绑定器：

```
ModelBinders.Binders.DefaultBinder = new MyNewDefaultBinder();
```

但是，修改默认绑定器会对应用程序的行为产生较大影响，因此在决定修改之前需要仔细斟酌。

3.3.2　DateTime 模型绑定器示例

在输入表单中，用户键入日期是常有的事。有时你还可以使用 jQuery 用户界面，让用户从图形日历中选择日期。如果在最新的浏览器上使用 HTML5 标记，日历将自动提供。所选择的日期会转换成一个字符串，保存到一个文本框中。当表单回传时，日期字符串被上传，默认绑定器会试图将其解析成 DateTime 对象。

在其他情形中，你可能会决定将日期拆分成三个不同的文本框，分别为日、月和年。它们分别作为请求中的不同值上传。其结果是默认绑定器只能单独管理它们；从日、月和年的值中创建有效 DateTime 对象的任务由控制器承担。有了自定义的默认绑定器，就可以使用控

制器中的代码，同时享受拥有控制器方法以下签名的乐趣：

```
public ActionResult MakeReservation(DateTime theDate)
```

我们看看如何设置一个更接近现实的模型绑定器的示例。

1. 显示数据

我们接下来要考虑的示例视图显示了组成日期项的三个文本框，以及一个提交按钮。你输入一个日期，系统会计算已经过了多少天或还有多少天才到你指定的那一天。下面是 Razor 标记：

```
@model DateEditorResponseViewModel
@section title{
    @Model.Title
}

@using (Html.BeginForm())
{
<fieldset>
    <legend>Date Editor</legend>
    <div>
        <table><tr>
        <td>@DateHelpers.InputDate("theDate", Model.DefaultDate)</td>
        <td><input type="submit" value="Find out more" /></td>
        </tr></table>
    </div>
</fieldset>
}
<hr />
@DateHelpers.Distance(Model.TimeToToday)
```

可以看到，我使用了两个自定义帮助器来更好地封装一些视图代码的呈现。下面是如何呈现日期元素的代码：

```
@helper InputDate(String name, DateTime? theDate)
{
    String day="", month="", year="";
    if(theDate.HasValue)
    {
        day = theDate.Value.Day.ToString();
        month = theDate.Value.Month.ToString();
        year = theDate.Value.Year.ToString();
    }
```

```
    <table cellpadding="0">
        <thead>
            <th>DD</th>
            <th>MM</th>
            <th>YYYY</th>
        </thead>
        <tr>
            <td><input type="number" name="@(name + ".day")"
                       value="@day" style="width:30px" /></td>
            <td><input type="number" name="@(name + ".month")"
                       value="@month" style="width:30px"></td>
            <td><input type="number" name="@(name + ".year")"
                       value="@year" style="width:40px" /></td>
        </tr>
    </table>
}
```

图 3-7 显示了输出结果。

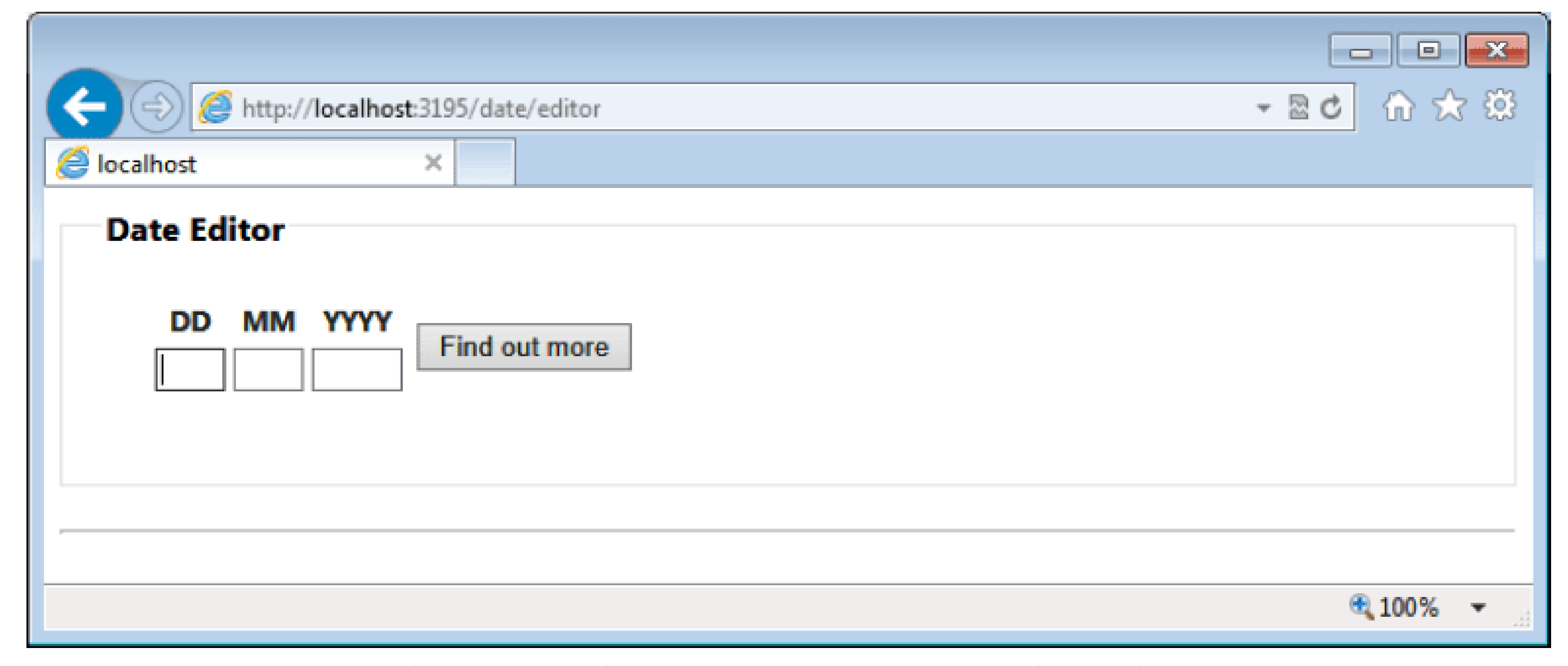


图 3-7　一个将日期输入文本拆分为日-月-年元素的示例视图

2. 控制器方法

图 3-7 中的视图由下面的控制器方法提供服务和处理：

```
public class DateController : Controller
{
    [HttpGet]
    [ActionName("Editor")]
    public ActionResult EditorForGet()
    {
        var model = new EditorViewModel();
        return View(model);
    }
```

```
    [HttpPost]
    [ActionName("Editor")]
    public ActionResult EditorForPost(DateTime theDate)
    {
        var model = new EditorViewModel();
        if (theDate != default(DateTime))
        {
            model.DefaultDate = theDate;
            model.TimeToToday = DateTime.Today.Subtract(theDate);
        }
        return View(model);
    }
}
```

日期回传后，控制器操作会计算出与当前日期的差异，并通过使用一个 TimeSpan 对象将结果存储在视图模型中。下面是该视图模型对象：

```
public class EditorViewModel : ViewModelBase
{
    public EditorViewModel()
    {
        DefaultDate = null;
        TimeToToday = null;
    }
    public DateTime? DefaultDate { get; set; }
    public TimeSpan? TimeToToday { get; set; }
}
```

还待探究的是将三个不同的独立上传值转换成一个 DateTime 对象的执行代码。

3. 创建 DateTime 绑定器

DateTimeModelBinder 对象的结构与我前面描述过的框架没有太大区别。只是它是为 DateTime 类型量身定做的。

```
public class DateModelBinder : IModelBinder
{
    public Object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        if (bindingContext == null)
        {
            throw new ArgumentNullException("bindingContext");
        }
```

```
        // This will return a DateTime object
        var theDate = default(DateTime);

        // Try to read from posted data. xxx.Day|xxx.Month|xxx.Year is assumed.
        var day = FromPostedData<int>(bindingContext, "Day");
        var month = FromPostedData<int>(bindingContext, "Month");
        var year = FromPostedData<int>(bindingContext, "Year");

        return CreateDateOrDefault(year, month, day, theDate);
    }

    // Helper routines
    private static T FromPostedData<T>(ModelBindingContext context, String id)
    {
        if (String.IsNullOrEmpty(id))
            return default(T);

        // Get the value from any of the input collections
        var key = String.Format("{0}.{1}", context.ModelName, id);
        var result = context.ValueProvider.GetValue(key);
        if (result == null && context.FallbackToEmptyPrefix)
        {
            // Try without prefix
            result = context.ValueProvider.GetValue(id);
            if (result == null)
                return default(T);
        }

        // Set the state of the model property resulting from value
        context.ModelState.SetModelValue(id, result);

        // Return the value converted (if possible) to the target type
        T valueToReturn = default(T);
        try
        {
            valueToReturn = (T)result.ConvertTo(typeof(T));
        }
        catch
        {
        }

       return valueToReturn;
    }
```

```
        private DateTime CreateDateOrDefault(Int32 year, Int32 month, Int32 day,
                                             DateTime? defaultDate)
        {
            var theDate = defaultDate ?? default(DateTime);
            try
            {
                theDate = new DateTime(year, month, day);
            }
            catch (ArgumentOutOfRangeException e)
            {
            }

            return theDate;
        }
    }
```

绑定器对这三个输入元素的命名规则做了一些假设。尤其是，它需要这些元素被命名为日、月和年，并可能以模型的名称为前缀。这是对前缀的支持，以便在同一视图中有多个日期输入框，且不发生冲突。

要使自定义绑定器可用，你只需要对它进行全局或局部注册。下面展示了如何使用一种特定的控制器方法来实现注册：

```
[HttpPost]
[ActionName("Editor")]
public ActionResult EditorForPost([ModelBinder(typeof(DateModelBinder))]
    DateTime theDate)
{
}
```

图 3-8 显示了实际运行中的最终页面。

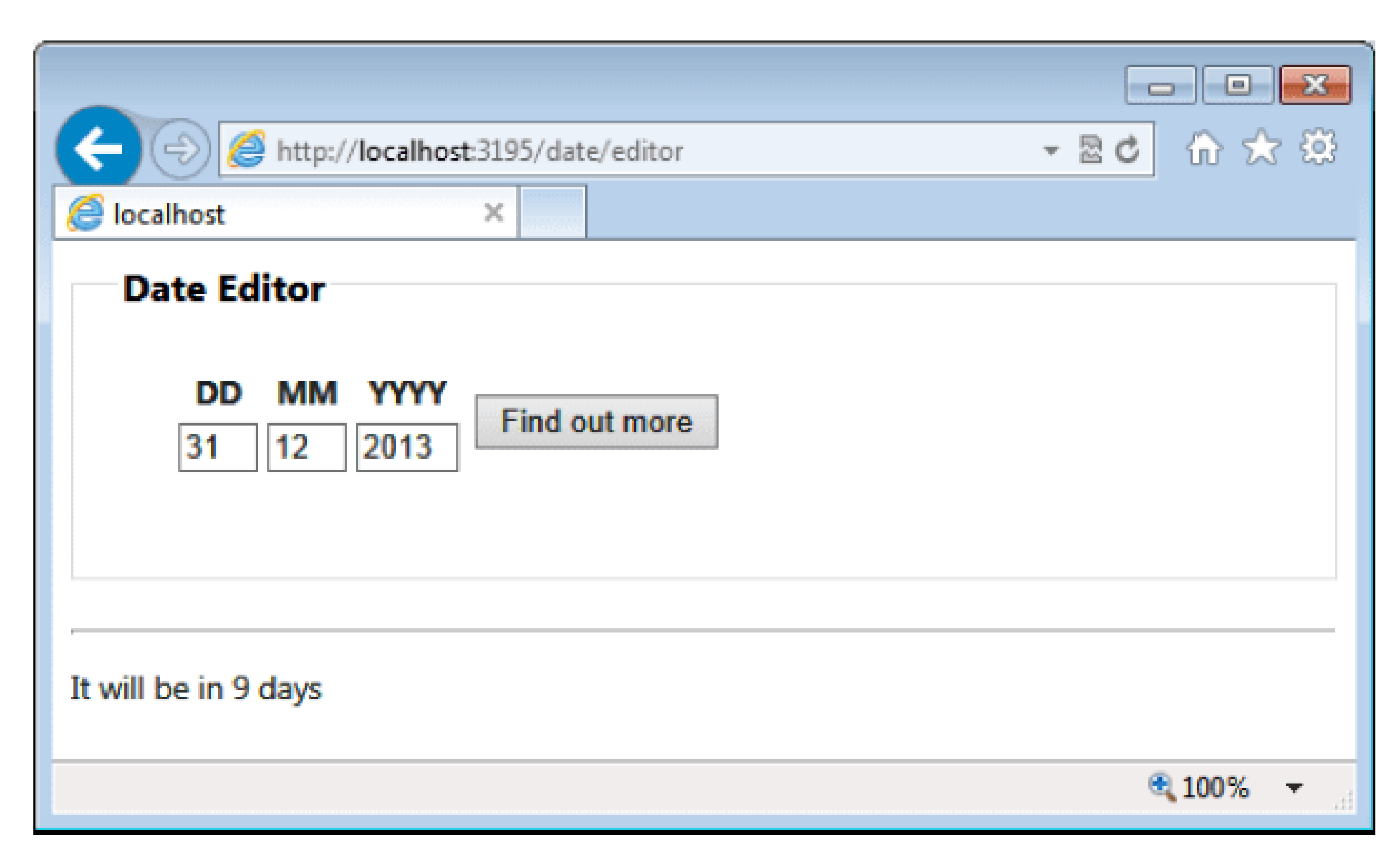


图 3-8　使用自定义类型绑定器处理日期

3.4　本章小结

在 ASP.NET MVC 和 ASP.NET Web Forms 中，提交的数据会进入 HTTP 数据包中一并提交，并映射到 Request 对象上的各种集合。为了向开发人员提供良好的服务，ASP.NET 随后会尝试以更有效的方式公开这些内容。

在 ASP.NET Web Forms 中，内容被解析并传递给服务器控件；而在 ASP.NET MVC 中，内容被绑定到所选控制器方法的参数。将提交值绑定到参数的过程称为模型绑定，是通过一个已注册的模型绑定器类实现的。模型绑定器为你提供了将表单提交值反序列化为简单类型和复杂类型的完全控制权。

在功能方面，默认绑定器的使用是对开发人员透明的——无须进行终端操作——并且它保持了控制器代码的整洁。通过使用模型绑定器，包括自定义绑定器，使控制器代码摆脱了对 ASP.NET 内在对象的依赖，因而使其变得更整洁和易于测试。

模型绑定器的使用与提交和输入表单紧密相关。在第 4 章中，将讨论输入表单、输入建模和数据验证几个方面的内容。

第 4 章

输 入 表 单

不管你能做什么，或者梦想做成什么，开始去做吧。

——沃尔夫冈 • 歌德

经典 ASP.NET 的编程模型是基于在回传间保持状态的假设的。在 HTTP 协议级别却并非如此，但是通过使用页面视图状态功能和针对 Web Forms 页面生命周期进行一些处理，就可以出色地模拟出这个假设。视图状态，这个大家往往避而远之的东西，却是在 ASP.NET 中建立有状态编程模型的大功臣，而该编程模型是 ASP.NET 取得成功且迅速普及的一个关键。数据输入是服务器控件真正发光发热的领域，它们的回传和视图状态系统开销使你避免了大量的工作。服务器控件也为你提供了一个强大的用于输入验证的基础架构。

如果对 Web Forms 及其服务器控件已经非常熟悉，那么当你转向 ASP.NET MVC 模型时你可能会感到震惊。在 ASP.NET MVC 中，与 Web Forms 相同的功能是要借助不同的工具集来完成的。ASP.NET MVC 框架使用不同的模式，它并不是以页面为基础的，而且依赖一个比 Web Forms 更单薄的抽象层。其结果是，你不会有丰富的如服务器控件这样的原生组件来快速配置一个友好的用户界面，其中元素可以在回传期间保留它们的内容。这一事实看似导致了生产效率的降低，至少对某些类型的应用程序来说是这样的，比如那些很大程度上基于数据输入的应用程序。

然而，事实真是这样吗？

确实，在 ASP.NET MVC 中你编写的代码在概念和实质上更接近于底层；因此，它用到了更多代码行，但它使你对生成的 HTML 和运行时环境的实际行为有了更多的控制。不过，你不需要一切从头开始编写。你有 HTML 帮助器，可以自动为任何简单或复杂的类型创建简单而实用的视图和编辑器。你有数据批注，可以用声明方式设置你对某字段的内容及其显示行为的预期。你有模型绑定器，可以将提交的值序列化为对服务器端的处理更为适用的对象。另外，你还有用于服务器和客户端验证的工具。

本章旨在展示如何通过 ASP.NET MVC 中的表单获得输入数据，然后在一个数据持久层对其进行验证和处理。

注意：

用于输入表单的基于 Ajax 的解决方案日益受到人们的重视。例如，向用户显示一个模态对话框，以便用户在其中键入输入数据这一情形变得越来越常见。显示一个模态对话框与创建一个单独的<div>一样简单，并有像 Twitter 引导程序或 jQuery UI 这样的客户端框架来管理呈现形式。但是就提交数据而言，使用普通的 FORM 提交以及本章所讨论的 ASP.NET MVC API 控制下的全部或部分页面刷新仍然是普遍的选择。另一个选项可能会使本章内容稍显无趣，那就是通过使用纯 JavaScript 代码自己管理 HTTP POST。

4.1　数据输入的一般模式

输入表单围绕两个主要模式：编辑-提交模式(Edit-and-Post)和选择-编辑-提交模式(Select-Edit-Post)。前者会显示一个 HTML 表单，且预期用户来填充字段并在完成时提交数据。后一种模式是对前者的扩展，是通过添加一个额外的预备步骤来实现的。用户选择一个数据项、将其置于编辑模式、编辑内容，然后将更改保存到存储层。

本章并未详细介绍编辑-提交(Edit-and-Post)模式，因为它只是选择-编辑-提交模式的简易版本。本章中的“编辑数据”和“保存数据”两节会对选择-编辑-提交模式进行介绍，并且提供对编辑-提交模式的描述。我们仍然用示例进行说明。

4.1.1　一个经典的选择-编辑-提交场景

将通过一个让用户从下拉列表选择客户的示例开始对选择-编辑-提交模式进行阐述。接着，包含所选客户信息的记录会呈现到编辑表单，可以在那里输入更新并最终验证和保存它们。

在这个示例中，域模型包含一个从经典的 Northwind 数据库推导而来的实体框架模型。图 4-1 显示了示例应用程序的初始用户界面。

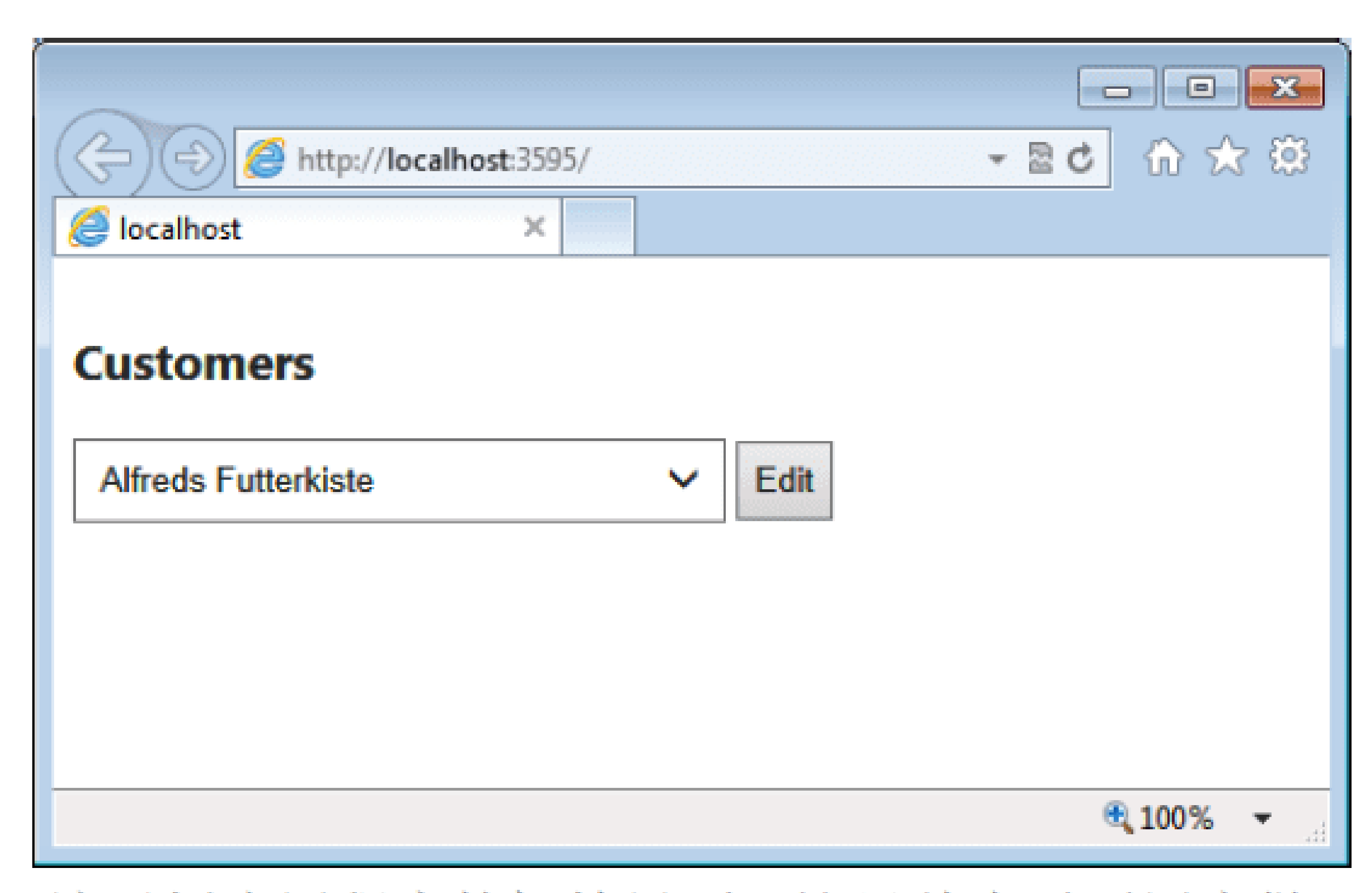


图 4-1　该示例应用程序的初始界面，这里首先需要用户做一个选择

注意：

本章中讨论的示例并没有在项目中使用任何以某种方式链接或嵌入的数据库。所有示例最终都把简单的查询调用指向到一个 Web 服务，这可以在我的个人网站 http://www.expoware.org 中找到。有关这些示例的详细信息，可以在 BookSamples.Components 库项目的 northwind.cs 文件中找到。

1. 提供数据和处理所选项

下面的列表显示了填充下拉列表以便为用户提供初始界面的控制器操作。注意我是根据职责驱动设计(Responsibility Driven Design，RDD)方法论中定义的协调器原型来设计控制器的(我还将在第 7 章“设计 ASP.NET MVC 控制器的注意事项”中讨论作为协调器的 RDD 和 MVC 控制器。现在姑且认为，协调器是一个对事件进行反应和将任何进一步的操作委托给外部组件的这样一种组件)。协调器仅限于传递输入和捕获输出。在 ASP.NET MVC 中将控制器作为协调器来实现是有益的，因为它可以把接收请求的层从处理请求的层中解耦出来。为简单起见，这里的代码并不使用依赖性注入。下一个显而易见的步骤是通过控制反转(Inversion of Control，IoC)来注入协调器类的实例——HomeService。

```
public class HomeController : Controller
{
    private readonly HomeService _service = new HomeService();
    public ActionResult Index()
    {
        var model = _service.GetModelForIndex();
        return View(model);
    }
}
```

Index 方法会获取一个视图模型对象，它包含了要显示的客户列表。

```
public class IndexViewModel : ViewModelBase
{
    public IndexViewModel()
    {
        Customers = new List<SimpleCustomer>();
    }
    public IEnumerable<SimpleCustomer> Customers { get; set; }
}
```

下面是协调器组件中 GetModelForIndex 方法的实现：

```
public class HomeService
{
```

```
        public IndexViewModel GetModelForIndex()
        {
            var model = new IndexViewModel {Customers =
                                               NorthwindCustomers.GetAll()};
            return model;
        }
    }
```

图 4-1 中产生接口的视图如下所示：

```
@model Sep.ViewModels.Home.IndexViewModel

<div class="floating">
    <p class="legend">Customers</p>
    @using(Html.BeginForm("edit", "home"))
    {
        @Html.DropDownList("customerList", new SelectList(Model.Customers,
                                "Id", "Company"))
        <input type="submit" name="btnEdit" value="Edit" />
    }
</div>
```

用户从列表中选择客户后(通过单击提交按钮)，即提交了一个 HomeController 类上的用于 Edit 操作的 POST 请求。

2. 编辑数据

用于 Edit 操作的请求会把应用程序转成编辑模式，并显示一个用于所选客户的编辑器。可在图 4-2 中看到，也应该预计到这样的连续视图可以保留下拉列表的当前状态。下面的代码显示了 Home 控制器上用于 Edit 方法的一个可能实现：

```
public ActionResult Edit([Bind(Prefix = "customerList")] String customerId)
{
    var model = _service.GetModelForEdit(customerId);
    return View(model);
}
```

第 3 章“模型绑定架构”指出，Bind 特性会指示默认模型绑定器将名为 customerList 的提交字段值分配到指定的参数。这种情况下，GetModelForEdit 方法会检索指定客户的信息，并传递给视图引擎，以便可以配置并向用户显示一个输入表单：

```
public EditViewModel GetModelForEdit(String id)
{
    var model = new EditViewModel
```

```
                    {
                        Title = "Edit customer",
                        Customer = NorthwindCustomers.Get(id),
                        Customers = NorthwindCustomers.GetAll()
                    };
    return model;
}
```

为何 GetModelForEdit 方法还需要检索客户列表？

这是没有视图状态的直接后果。视图的所有内容每次都必须重新创建和重新填充。这是 HTTP 协议的一个关键部分，涉及固有的 HTTP 无状态特性。在 ASP.NET Web Forms 中，大部分的再填充工作都是由框架的抽象层通过存储在视图状态中的信息自动完成的。而在 ASP.NET MVC 中，这些都需要你来完成。

在前面的代码片段中，设置了另一个到服务层的调用；一个更重要的应用程序可能会缓存数据，并从那里重新加载。但是，缓存层也可以纳入存储库本身。

下面是视图的代码：

```
@model Sep.ViewModels.Home.EditViewModel

<div class="container">
    <div class="floating">
        <p class="legend">Customers</p>
        @using(Html.BeginForm("edit", "home"))
        {
            @Html.DropDownList("customerList",
                new SelectList(Model.Customers, "Id", "Company",
                     Model.Customer.Id))
            <input type="submit" name="btnEdit" value="Edit" />
        }
    </div>
    <div class="floating-spacer">
        @Html.Partial("uc_customerEditor", Model.Customer)
    </div>
</div>
```

视图的结构与图 4-1 所示的差不多相同，唯一的区别在于视图右侧是基于表格的编辑器。该编辑器(uc_customerEditor.cshtml)是通过 Partial HTML 帮助器创建的。如果仔细观察原生 HTML 帮助器的列表，会发现两个看似相似的帮助器：Partial 和 RenderPartial。它们的区别是什么呢？第 2 章“ASP.NET MVC 视图”中有提示，Partial 只返回字符串，而 RenderPartial 会执行呈现字符串的操作。如果目标只是创建视图，那么它们几乎完全相同，但仍然需要一种略微不同的编程语法。要调用 RenderPartial，你需要在 Razor 中定义如下标记：

```
@{ Html.RenderPartial(view) }
```

图 4-2 显示了实际运行中的编辑器。

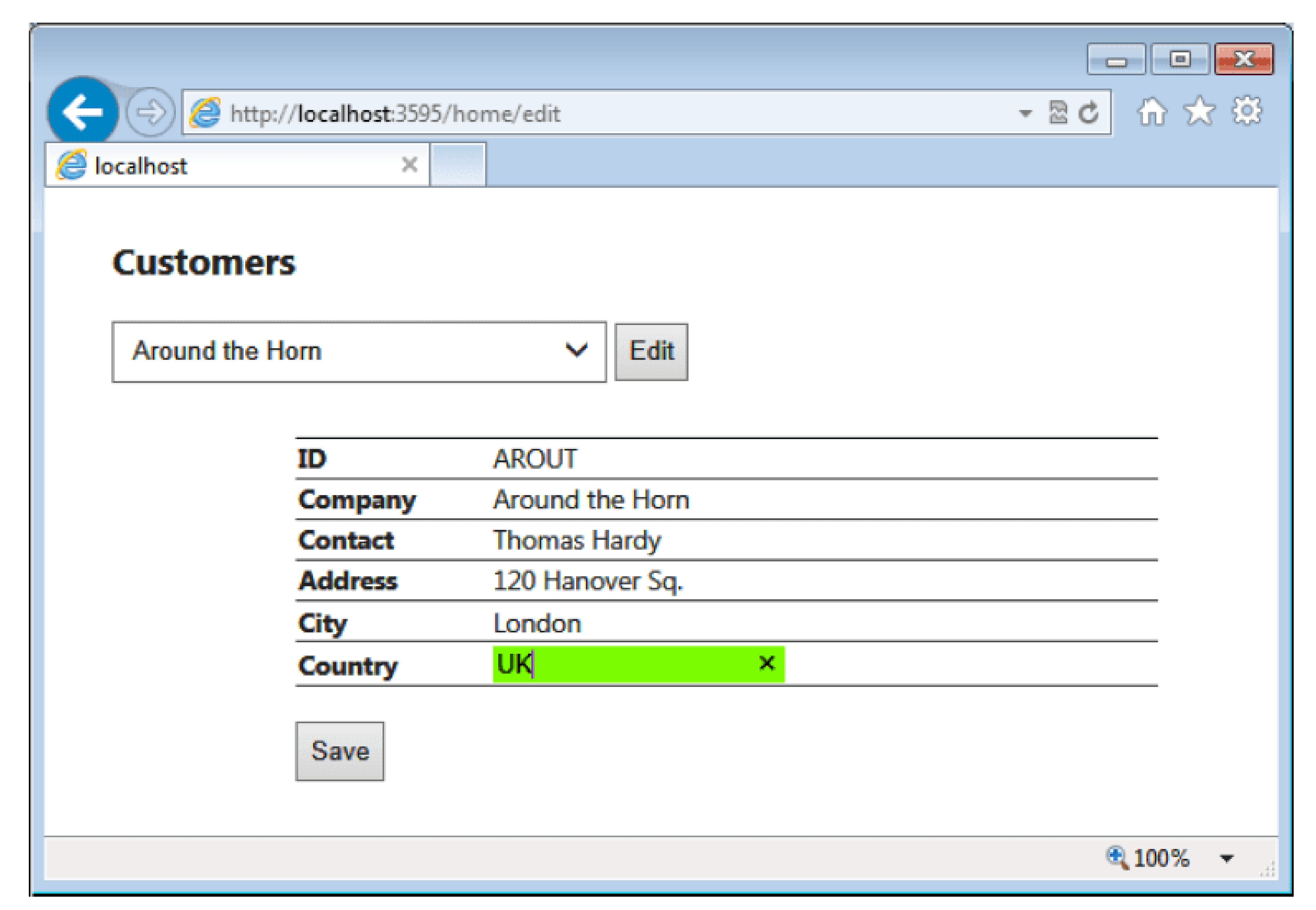


图 4-2　用户可以修改选中的客户

3. 保存数据

在显示输入表单之后，用户可输入他认为有效的任何数据，然后按下按钮将表单内容提交到服务器。下面是提交变更的一个典型表单标记(该标记是 uc_customerEditor.cshtml 文件的内容)。

```
@model BookSamples.Components.Data.SimpleCustomer

<div class="container">
@using(Html.BeginForm("update", "customer", new { customerId = Model.Id }))
{
   <table id="__tableCustomerEditAscx" rules="rows" frame="hsides">
      <tr>
         <td width="100px"><b>ID</b></td>
         <td width="350px">@Model.Id</td>
      </tr>
      <tr>
         <td><b>Company</b></td>
         <td><span>@Model.Company</span></td>
      </tr>
      <tr>
         <td><b>Contact</b></td>
```

```
            <td>@Html.DisplayTextFor(c => c.Contact)</td>
        </tr>
            ...
        <tr>
            <td><b>Country</b></td>
            <td>@Html.TextBoxFor(c => c.Country,
                      new Dictionary<String, Object>() { { "class",
                       "textBox" } })
                <br />
                @Html.ValidationMessage("country")
            </td>
        </tr>
    </table>
    <br/>
    <input id="btnSave" type="submit" value="Save" />
}
</div>
```

通常情况下，你要用一个 ValidationMessage 帮助器对所有可编辑的字段(比如文本框)进行分组。验证帮助器会显示由于字段中输入无效值而产生的任何结果。此外，你要确保生成的 URL 包含一个关键值，用于要更新记录的唯一标识。下面是一个示例：

```
Html.BeginForm("update", "home", new { customerId = Model.CustomerID })
```

在内部，BeginForm 会将它接收的数据与已注册 URL 的路由参数相匹配，试图创建合适的 URL 以提交表单。前面的代码会生成下面的 URL：

```
http://yourserver/customer/update?customerId=alfki
```

正是由于有了默认的模型绑定器，所有 Update 方法才可以作为 SimpleCustomer 类的成员接收输入表单的字段：

```
public ActionResult Update(SimpleCustomer customer)
{
    var modelState = ViewData.ModelState
    var model = _service.TryUpdateCustomer(modelState, customer);
    return View("edit", model);
}
```

该方法需要做两件事：更新数据层和显示编辑视图，这样用户就可以持续进行更改。除了一些异常简单的场景，更新操作都需要进行验证。如果对正在被存储的数据的验证失败，那么检测到的错误必须通过用户界面报告给最终用户。

ASP.NET MVC 的基础架构对源于验证的错误消息的显示提供了内置支持。ModelState

字典——Controller 类的一部分——是方法添加错误通知的地方。ModelState 字典中的错误随后会通过 ValidationMessage 帮助器显示出来，如下所示：

```
public EditCustomerViewModel TryUpdateCustomer(ModelStateDictionary
      modelState, Customer customer)
{
    if (Validate(modelState, customer))
        Update(customer);

    return EditCustomer(customer.CustomerID);
}

private static Boolean Validate(ModelStateDictionary modelState, Customer
      customer)
{
    var result = true;

    // Any sort of specific validation you need ...
    if (!CountryIsValid(customer.Country))
    {
        // For each detected error, add a message and set a new display value
        modelState.AddModelError("Country", "Invalid country.");
        result = false;
    }

    return result;
}

private static void Update(Customer customer)
{
    NorthwindCustomers.Update(customer);
}
```

顾名思义，ModelState 字典就是与视图背后的模型状态有关的所有消息的关键字/值存储库。其中值就是错误消息；关键字是用于标识条目(如前面示例中的字符串“Country”)的唯一名称。模型状态条目的关键字要与 Html.ValidationMessage 帮助器的字符串参数相匹配。图 4-3 显示了当用户输入无效值时系统的反应。

验证过程中，每当在提交的数据中检测出一个错误，就会在 ModelState 字典中添加一个新条目，如下所示：

```
modelState.AddModelError("Country", "Invalid country.");
```

提供本地化的错误消息是你的责任。

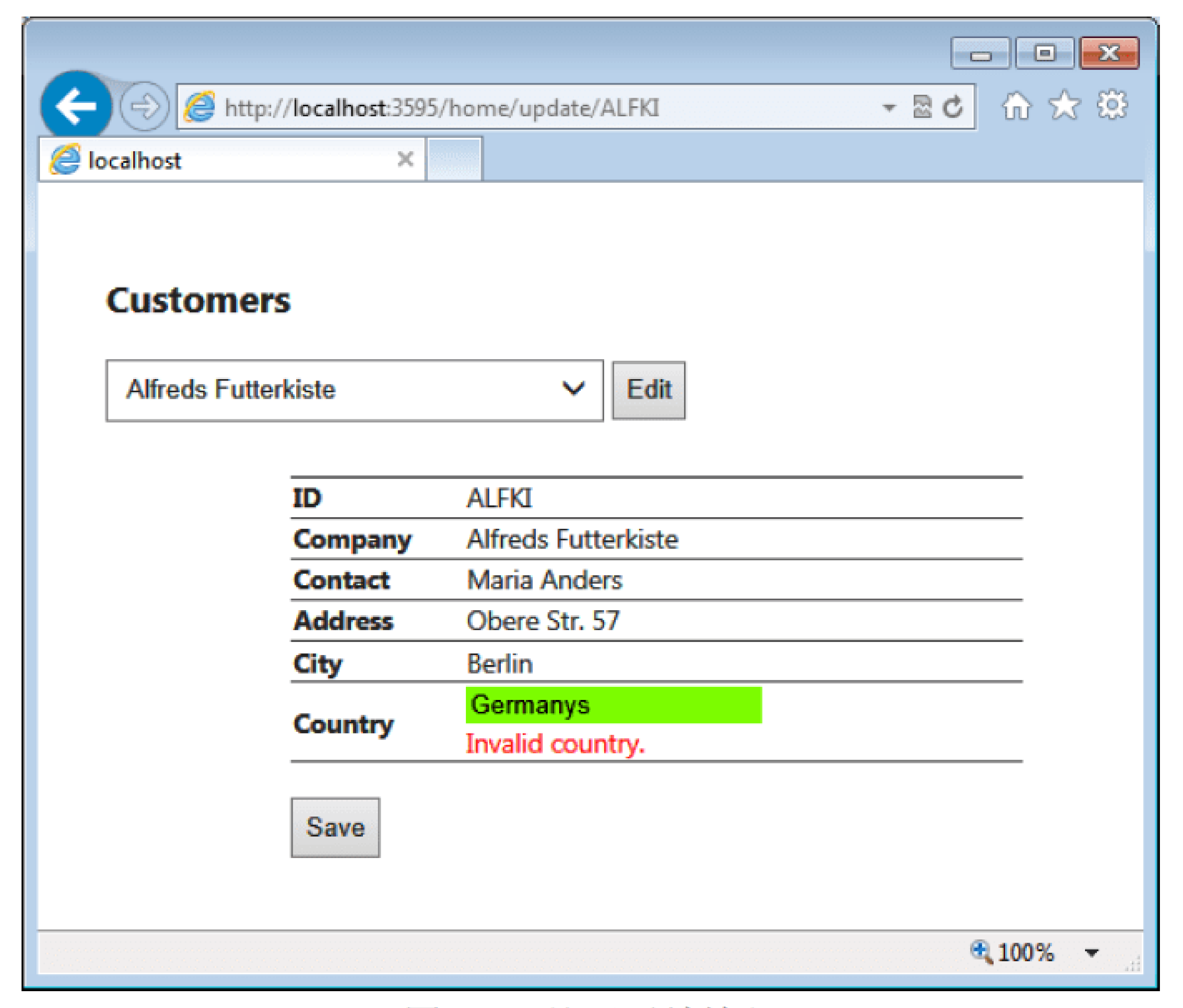


图 4-3　处理无效输入

注意，如果不能通过各种值提供程序(比如表单、路由参数或查询字符串)找到相应的匹配，则模型绑定器会将参数设置为 null。尤其，这意味着字符串参数或字符串成员可以被设置为 null。在尝试使用通过了模型绑定的值之前，你应该随时检查是否有 null 值。下面的代码显示了当涉及字符串时，处理这种情况的可行方法：

```
// In the examples for this chapter, validation is really coding in this way.
   In the end,
// validation is a plain logical expression suggested by business. While this
   condition
// is not much realistic, it still makes perfectly sense in a demo. Anyway,
   note that in light
// of this code any "valid" country name you enter is denied except "USA".
if (String.IsNullOrEmpty(customer.Country)
   || !customer.Country.Equals("USA"))
{
   ...
}
```

考虑前面的代码，要在 CountryIsValid 方法中放置针对 null 值的检验。

4.1.2　应用提交-重定向-获取(Post-Redirect-Get)模式

以前输入表单的方法很实用，但并不完美。首先，地址栏中的 URL 不一定反映页面显示

的数据。其次，重复上次的操作(刷新或 F5 按键)可能不只是提示用户如图 4-4 所示的烦人的确认消息。随着用户继续操作，如果重复的操作没有幂等性实现(即：不论被连续调用多少次，都没有产生相同的结果)，便很可能引发运行时异常。

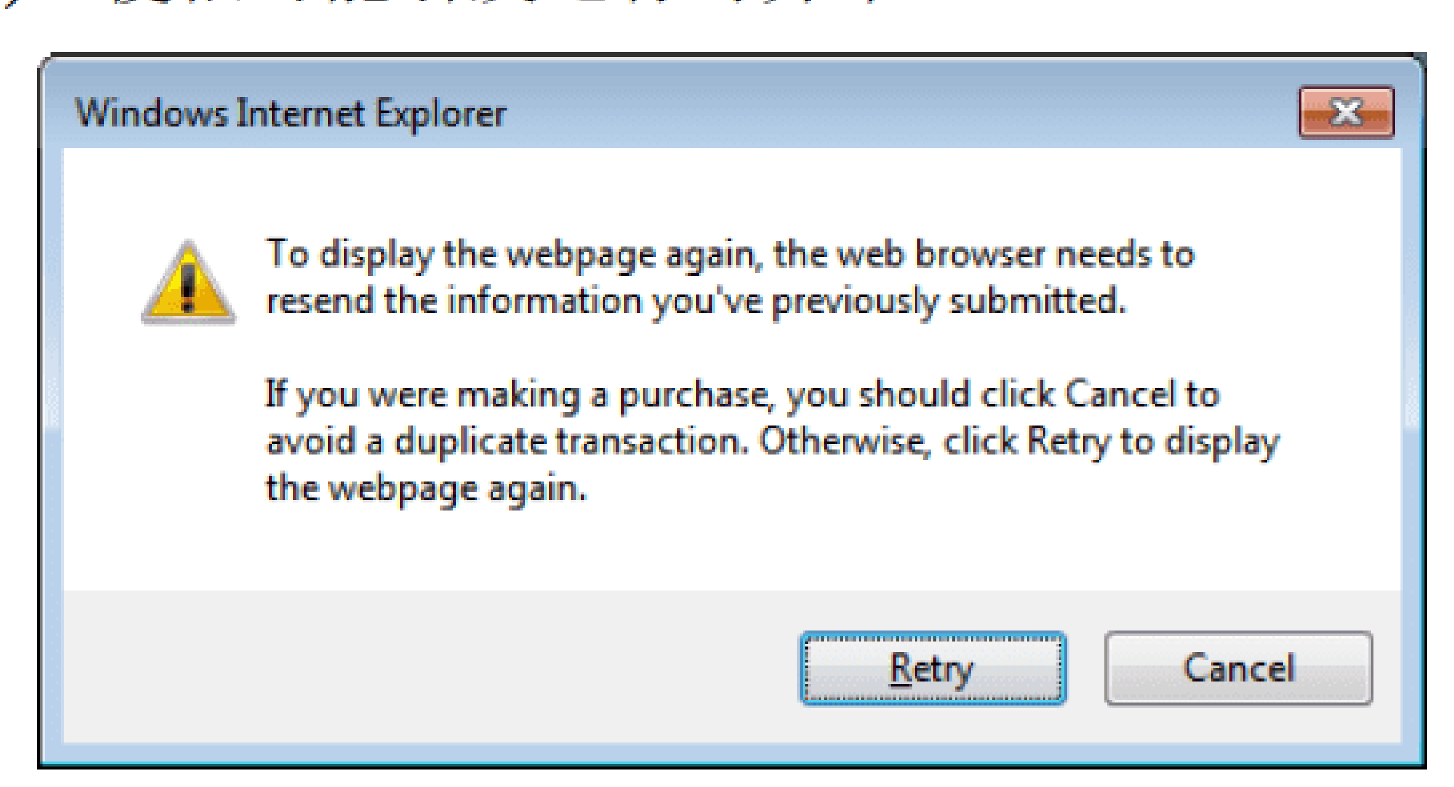


图 4-4　重新提交表单时浏览器显示的确认消息

然而，ASP.NET 应用程序的这些方面并不是特别针对 ASP.NET MVC 的。它们也存在于 ASP.NET Web Forms 中，但这并非避免使用更好实现方法的理由。

浏览器地址栏中的 URL 与显示的内容之间缺乏同步性在大多数情况下不算什么问题。甚至你的用户都不一定会注意到。事实上，当用户刷新当前页面时提示用户的确认对话框对于 ASP.NET 开发人员以及他们的应用程序用户来说并不陌生(也并不愉快)。我们看看提交-重定向-获取(PRG)模式如何帮助解决这两方面的问题。

1. 保持 URL 与内容同步

比如，在前面的示例中，当用户选择用户第一条记录(ALFKI)之后，地址栏中会显示如下的 URL：

```
// Action EDIT on CUSTOMER controller
http://yourserver/customer/edit
```

如果用户重复上次的操作(比如，通过按 F5)，会得到如图 4-4 所示的对话框，之后视图会如预期一样更新。用 URL 来反映所选的客户以及无副作用地使页面刷新，这样不是很好吗？

由图 4-4 中对话框所代表的副作用有一个著名的起源。当按下 F5 键时，浏览器只是盲目地重复进行最后一个 HTTP 请求。并且，客户从下拉列表中所做的选择(参见图 4-1)是 Customer 控制器上 Edit 操作的一个 HTTP POST 请求。

PRG 模式建议每个 POST 请求在经过处理后，以重定向到一个通过 GET 访问的资源而结束。这样做能够使 URL 与所显示的客户保持良好的同步，并且用户不会再看到图 4-4 中那令他们讨厌的对话框。

2. 分隔 POST 和 GET 操作

将 PRG 应用到 ASP.NET MVC 应用程序的第一步是将 POST 操作与 GET 操作清晰地分隔开来。下面是如何重写 Edit 操作：

```
[HttpPost]
[ActionName("edit")]
public ActionResult EditViaPost([Bind(Prefix = "customerList")] String
  customerId)
{
    // POST, now REDIRECT via GET to Edit
    return RedirectToAction("edit", new {id = customerId});
}

[HttpGet]
[ActionName("edit")]
public ActionResult EditViaGet(String id)
{
    var model = _service.GetModelForEdit(id);
    return View("edit", model);
}
```

每一次用户提交 Edit 方法以选择一个指定客户时，所发生的都是一个通过 GET 到相同操作的重定向(HTTP 302)。用于 Edit 操作的 GET 方法会接收要编辑的客户 ID，并像平常一样工作。

不错的效果是可以用两种方式改变选择：在地址栏中键入 URL(作为命令)或仅仅单击下拉列表。此外，当用户界面更新了，由浏览器跟踪的最后一个操作是 Get，那么就可以按自己的意愿多次重复这一 Get 操作，而不会招致任何无聊的警告或烦人的异常。

3. 仅通过 POST 更新

回头再看看图 4-3，它显示了更新后的页面。特别是图中显示了一个失败的更新，但与此处要介绍的并不相关。相反，相关的是其 URL。

```
http://yourserver/customer/update?customerId=ALFKI
```

这是按 F5 键时会重复的 URL。很难相信任何普通用户会尝试手动编辑此 URL，并试图将更新推送给另一个客户。然而可能性小归小，但绝对是有可能的。

更新操作的 URL 不应该对用户可见。用户会执行更新，但操作仍然隐藏在同一页面的两次显示之间。这与 PRG 模式中的情况完全一样。下面是如何重写 Update 操作的代码：

```
[HttpPost]
```

```
public ActionResult Update(SimpleCustomer customer)
{
     var modelState = ViewData.ModelState;
     _service.TryUpdateCustomer(modelState, TempData, customer);
     return RedirectToAction("edit", new { id = customer.Id });
}
```

可以看出，你需要的只是 HttpPost 支撑。用户会从一个通过 GET 操作创建的页面执行更新，该页面显示了正在被编辑的客户。更新随即发生，接下来的视图是通过再次重定向到相同客户的 Edit 操作来获得的。这一过程十分简单、简洁和高效。

4. 跨重定向的临时数据保存

还有最后一个问题需要考虑。PRG 模式使整个代码看起来更整洁，但需要两次请求以便更新视图。这可能造成性能方面的问题，但也说不定——我认为可能性不大。无论有没有，都主要是指功能上的问题：如果更新失败，你如何将反馈传递给视图？事实上，在 GET 操作中视图会继重定向之后进行呈现；它是一个截然不同的空白请求。

总体而言，最好的选择是将反馈消息保存到 Session 对象。ASP.NET MVC 提供了一个略微好一点的选项，该选项仍然需要使用会话状态，不过是通过智能封装——TempData 字典实现的。下面演示了如何在更新之前修改验证代码：

```
private static Boolean Validate(ModelStateDictionary modelState,
                                TempDataDictionary tempData,
                                SimpleCustomer customer)
{
    var result = true;

    if (String.IsNullOrEmpty(customer.Country)
        || !customer.Country.Equals("USA"))
    {
        modelState.AddModelError("Country", "Invalid country.");

        // Save model-state to TempData
        tempData["ModelState"] = modelState;
        result = false;
    }

    return result;
}
```

首先像往常一样将反馈消息添加到 ModelState 字典。然后，将对 ModelState 的引用保存到 TempData 字典。TempData 字典会在两次请求期间把你提供的所有数据保存在会话状态中。

在通过存储器的第二个请求被处理后，容器将清除该条目。

对于其他任何你需要传递到视图对象的信息或数据都要做相同的处理。例如，图 4-5 描述了在更新操作成功完成以后通过下列代码添加的信息：

```
private Boolean Update(TempDataDictionary tempData, SimpleCustomer customer)
{
    // Perform physical update
    var result = _repository.Update(customer);

    // Add a message for the user
    var msg = result
                ? "Successfully updated."
                : "Update failed. Check your input data!";
    tempData["OutputMessage"] = msg;
    return result;
}
```

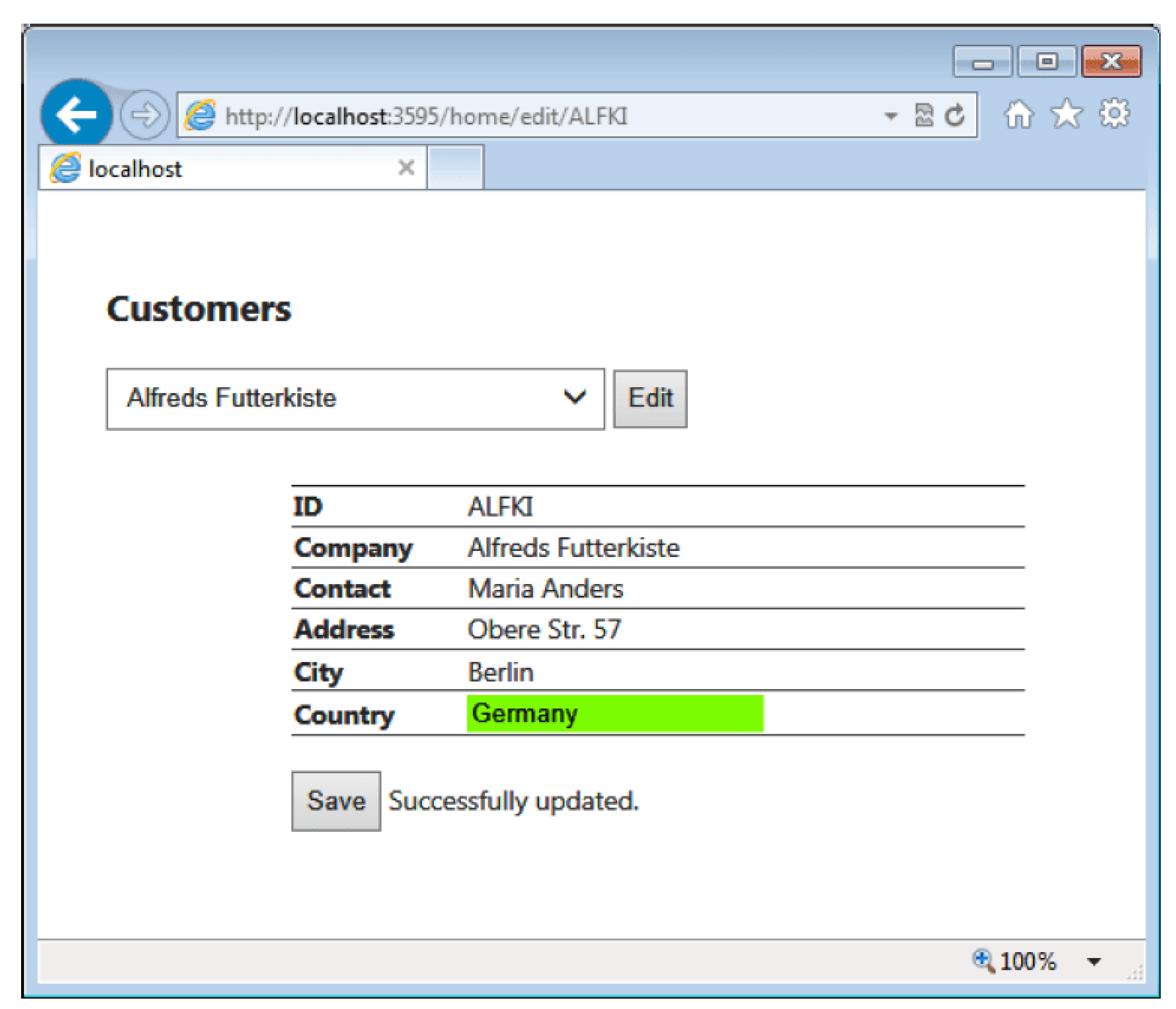


图 4-5　更新提交后的消息

你可能仍然会想在重新显示相同编辑表单之前就所发生的事情向用户提供一些明确的反馈，以使用户继续正常的工作。如果更新操作把用户带到了完全不同的页面，可能就不需

要前面的工作了。

将数据保存到 TempData 字典只完成了一半的工作。还需要添加从会话状态中检索字典的代码，并将其与当前的模型状态合并。这个逻辑要在实际呈现视图的代码中实现。

```
 [HttpGet]
[ActionName("edit")]
public ActionResult EditViaGet(String id)
{
   // Merge current ModelState with any being recovered from TempData
   LoadStateFromTempData();

    ...
}
private void LoadStateFromTempData()
{
    var modelState = TempData["ModelState"] as ModelStateDictionary;
    if (modelState != null)
        ModelState.Merge(modelState);
}
```

有了这些少许变化，就可以设置读写简单且运行高效的输入表单了。按照 PRG 模式构建的输入表单，其唯一缺陷是每个操作都需要请求二次重定向。此外，还需将带有呈现视图所需数据的视图模型封装起来，包括由正在进行的操作所计算出的数据以及与页面有关的任何其他数据，如菜单、面包屑导航和列表等。

不过，要超越这一实现水平，你需要采用 Ajax 方式。

超越经典的以浏览器为主导的内容表单提交

经典的表单提交大部分发生在整个页面刷新的情况下，有时候发生在两次 HTTP 请求的情况，就像 PRG 模式中一样。有没有无需整页刷新就可实现输入表单的方式呢？

确实有几种方式，但它们各有技巧，因此请以个人喜好为准。一般情况下，我会使用两种可能的方法构建自己的表单：经由 PRG 模式提交的普通旧式表单、和使用 Ajax 的模态输入表单。通过 Ajax.BeginForm HTML 帮助器，ASP.NET MVC 的确对 Ajax 驱动的提交有一些原生支持。另外，可通过使用一些技术来手动构建模态输入表单，如用于用户界面的 Twitter 引导程序(和模态弹出式基础架构)、用于客户端检验的 jQuery 验证以及用于提交的 XmlHttpRequest(XHR)。

http://software2cents.wordpress.com/2013/06/07/modal-input-forms-with-bootstrap/是我的一个博客帖子，在其中的 ASP.NET MVC 应用程序上下文中，可以找到一个不错的模态输入表单的启动示例。该帖子说明了如何使用集成于 ASP.NET MVC 5 的 CSS 库中的 Twitter 引导程序。

4.2　输入表单的自动化编写

输入表单在视图的组织结构中扮演了十分重要的角色。输入表单通常包含图片、请求字段的具体位置，以及丰富的客户端内容。在所有这些情况下，你很可能需要从头开始编写输入表单的模板。

然而，一般来说，有许多其他情况可以自动化构建输入表单。比如，编写一个网站的后台就属于这种情况。后台系统通常包括大量用于编辑记录的表单；但是，作为开发人员，你不用关心图表的问题。你需要关注的是效用，而不是样式。简而言之，后台系统是不用顾虑地使用表单输入模板的自动生成器的完美情形。

4.2.1　预定义的显示和编辑器模板

第 2 章展示了模板化的 HTML 帮助器，如 DisplayXxx 和 EditorXxx 等。这些帮助器可以使用一个对象(甚至传递到视图的整个模型)并构建一个只读或可编辑的表单。可以使用下面的表达式来指定要显示或编辑的对象：

```
Html.DisplayFor(model => model.Customer)
Html.EditorFor(model => model.Customer)
```

model 参数往往是要传递给视图的模型。可以使用自己的 lambda 表达式来选择整个模型的一个子集。要选择整个模型(比如说，为了进行编辑)，可以选择下面任意一个功能相当的表达式：

```
Html.EditorFor(model => model)
Html.EditorForModel()
```

置于模型类的公共成员上的特性提供了对如何显示单个值的指引。我们看看这是如何用于实践的。

1. 用于显示的批注数据成员

在 ASP.NET MVC 中，模板化帮助器使用与类成员相关联的元数据来决定如何显示或编辑你的数据。元数据通过元数据提供程序对象读取；默认的元数据提供程序会从数据批注特性中获取信息。表 4-1 列出了最常用的特性。

表 4-1　一些影响数据呈现的批注

特　　性	描　　述
DataType	表示你将通过该成员编辑的推测数据类型。它会接收来自 DataType 枚举的值。支持的数据类型包括 Decimal、Date、DateTime、EmailAddress、Password、Url、PhoneNumber 以及 MultilineText
DisplayFormat	用此表示通过哪种格式来显示(和/或编辑)值。例如，可使用此批注来表明一个 null 或空值的替代表示。此例中，你要使用 NullDisplayText 属性
DisplayName	表示用于表示值的标签的文本
HiddenInput	表示一个隐藏输入字段是否应该显示以替代一个可见输入字段
UIHint	表示在显式或编辑值时使用的自定义 HTML 模板的名称

批注存在于各种类型的名称空间中，包括System.ComponentModel和System.ComponentModel.DataAnnotations。如果探究这些(以及其他)名称空间，你甚至可以找到更多特性，但其中一些也许不能用于ASP.NET MVC，至少不会以你认为的方式。

数据批注是特性，而特性通常不包含代码。它们仅仅表示其他模块需要使用的元信息。通过使用数据批注，你的模型对象就被元数据修饰了。这并不能预计产生任何的可见影响：一切都取决于其他组件如何使用元数据。

在 ASP.NET MVC 中，默认的显示和编辑帮助器只使用几个可能的批注。然而，元数据信息就在那里，并且如果重写默认的模板，就可以在闲暇时使用更多可用的元信息了。ReadOnly 特性就是一个批注的好例子，它常常被默认模板忽略，但 ASP.NET MVC 能够解析它并将其公开给帮助器。

注意：

数据批注包括描述性特性和验证特性，描述性特性指示侦听器如何显示或编辑数据，验证特性指示侦听器如何验证模型类的内容。稍后将介绍验证特性。

下面的代码显示了一个用批注修饰的视图模型类：

```
public class CustomerViewModel : ViewModelBase
{
    [DisplayName("Company ID")]
    [ReadOnly(true)]  // This will be blissfully ignored by default templates!
    public Int32 Id { get; set; }

    [DisplayName("Is a Company (or individual)?")]
    public Boolean IsCompany { get; set; }
```

```
        [DisplayFormat(NullDisplayText = "(empty)")]
        public String Name { get; set; }

        [DataType(DataType.MultilineText)]
        public String Notes { get; set; }

        [DataType(DataType.Url)]
        public String Website { get; set; }

        [DisplayName("Does this customer pay regularly?")]
        public Boolean? IsReliable { get; set; }
    }
```

注意如果没有使用 DisplayForModel 或 EditorForModel 自动生成输入表单，就不需要数据批注。与任何其他元数据一样，批注对所有的非专门设计为使用它们的代码也是透明的。

图 4-6 显示了 EditorForModel 创建的用于前面模型对象的输入表单。

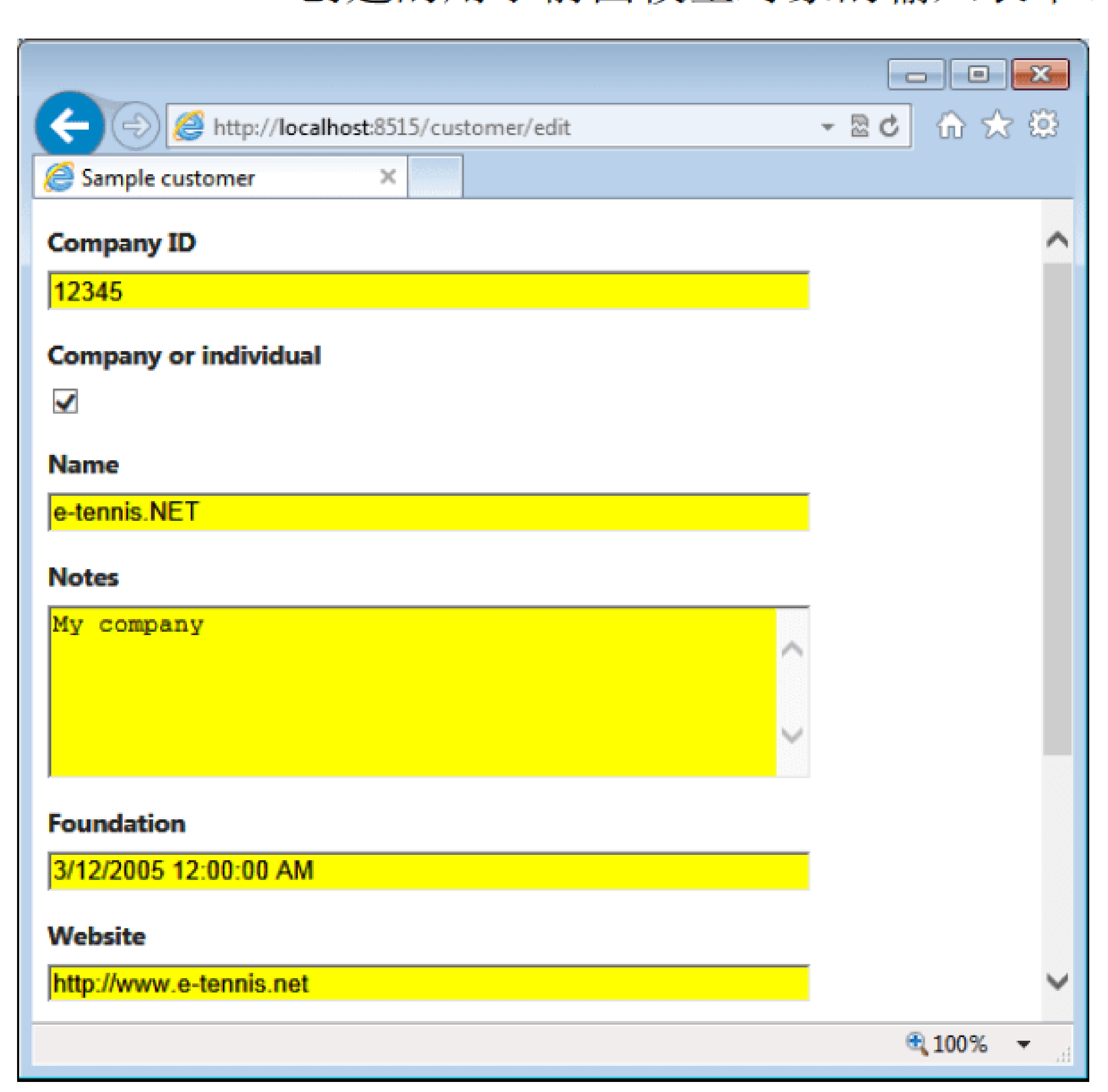


图 4-6　一个自动生成的输入表单

元信息会指示编辑/显示帮助器如何编辑和显示值。这会导致专有 HTML 模板的使用，如用于多行类型的 TextArea 元素，以及用于 Boolean 值的复选框。对于可为 null 的 Boolean 值，帮助器还会自动显示一个三态的下拉列表。并非所有数据类型都要在编辑模式和显示模式中显示。比如，EmailAddress 和 Url 数据类型仅仅在显示模式中表现。

确切地说，由于样式表的应用，图中显示的表单还要经历一组微小的图形变更。编辑/

显示帮助器会从带有惯用名称的如下所示的几个级联样式表(CSS)类中自动读取样式信息：

```
.display-label, .editor-label {
   margin: 1em 0 0 0;
   font-weight: bold;
}
.display-field, .editor-field {
   margin: 0.5em 0 0 0;
}
.text-box {
   width: 30em;
   background: yellow;
}
.text-box.multi-line {
   height: 6.5em;
}
.tri-state {
   width: 6em;
}
```

尤其是，名为 display-label 和 editor-label 的类指的是围绕输入值的标签。可以将这些样式内嵌到视图中，也可以从全局共享的 CSS 文件中继承它们。

2. 用于数据类型的默认模板

显示/编辑帮助器是通过映射每个成员的数据类型以呈现给预定义的显示或编辑模板来发生作用的。之后，对于每个模板名称，系统会要求视图引擎返回一个适当的部分视图。预定义的显示模板因为这些数据类型而存在：Boolean、Decimal、EmailAddress、HiddenInput、Html、Object、String、Text 和 Url。数据类型是通过 DataType 批注或值的实际类型来解析的。如果找不到匹配项，则使用默认模板，它包含用于显示的普通文本和用来编辑的文本框。让我们看看模板是什么样子。下面的清单展示了一个用于显示 Url 数据类型的 Razor 模板的实现示例：

```
@inherits System.Web.Mvc.WebViewPage<String>
<a href="@Model">
  @ViewData.TemplateInfo.FormattedModelValue
</a>
```

Url 数据类型通过一个超链接来呈现，其中的 Model 对象会引用要显示的数据——很有可能是一个表示网站的字符串。你会发现，该实际值是用来设置 URL 而不是超链接的文本。TemplateInfo 对象上的 FormattedModelValue 属性可以是最初的原始模型值，也可以是一个正确格式化的字符串，如果通过批注指定了格式字符串的话。

注意：

TemplateInfo 属性在 ViewData 对象上定义，但通常为 null，除非你正处于一个模板中。

Url 模板相当简单。让我们思考一个更有趣的 Boolean 值的模板吧：

```
@model bool?

@if (Model == null)
{
    <text>Not Set</text>
}
else
{
    if (Model.Value)
    {
        <text>True</text>
    }
    else
    {
        <text>False</text>
    }
}
```

由于该模板适用于 Boolean 值和 Nullable<Boolean>值，因此该示例包含了逻辑和标记的组合。

编辑模板与显示模板并没有太多不同，至少在结构上是差不多的；但是，编辑模板的代码会更复杂一些。ASP.NET MVC 提供了几个预定义的编辑器用于 Boolean、Decimal、HiddenInput、Object、String、Password 和 MultilineText。下面是一个输入密码的编辑器示例：

```
@Html.Password("",
        ViewData.TemplateInfo.FormattedModelValue,
        new { @class = "text-box single-line password" })
```

可以看到，有关样式的约定源于默认的模板实现。通过更改指定数据类型的默认模板，就可以根据不同的规则选择样式了。

Object.cshtml 是用于递归循环处理某类型公共成员的模板。它通过使用专用于类型的编辑器来构建整个表单。Object.cshtml 的默认实现会垂直堆叠标签和编辑器。虽然它对于快速建模较有价值，但你通常不会真正考虑将它用于实际应用程序。我们继续探究如何自定义编辑器模板吧。

3. 用于数据类型的自定义模板

显示/编辑帮助器在很大程度上可以自定义。任何自定义的模板都包含一个位于 Views\[controller]\DisplayTemplates 文件夹用于显示帮助器的自定义视图，和一个位于 Views\[controller]\EditorTemplates 文件夹用于编辑帮助器的自定义视图。如果希望模板为所有的控制器共享，就将它们放在 Views\Shared 中。如果视图的名称与数据类型相匹配，则该视图会变成这个数据类型的新默认模板。如果不能匹配，则该视图不会使用，除非通过 UIHint 批注显式调用，如下所示：

```
public CustomerViewModel
{
    ...
    [UIHint("CustomerViewModel_Url")]
    public String Url {get; set;}
}
```

现在 Url 属性通过使用 CustomerViewModel_Url 模板进行显示和编辑。而数据类型是 Url 的属性会继续由默认的模板服务。

我们看看对于那些甚至没有预定义模板的类型——DateTime 类型，如何创建一个自定义的显示和编辑器模板。下面是一个 Razor 显示模板的内容：

```
@model DateTime
@Model.ToString("ddd, dd MMM yyyy")
```

在该示例中，你用一个固定的格式(但由你控制)显示日期，包括星期几、几月份和哪一年。下面是一个用于日期的编辑器模板示例：

```
@model DateTime
<div>
<table>
<tr>
    <td>@Html.Label("Day")</td>
    <td>@Html.TextBox("Day", Model.Day)</td>
</tr>
<tr>
    <td>@Html.Label("Month")</td>
    <td>@Html.TextBox("Month", Model.Month)</td>
</tr>
<tr>
    <td>@Html.Label("Year")%></td>
    <td>@Html.TextBox("Year", Model.Year)</td>
</tr>
</table>
```

```
</div>
```

可通过使用任何你喜欢的类名在模板中设计元素样式；只要那些类是在应用程序中的某些样式表中定义的，它们就会被自动使用。注意你在模板中设置的 ID 会自动在发出标记中加上成员名称的前缀。例如，如果把上面的日期模板应用到一个名为 BirthDate 的属性，所发出的实际 ID 就会是 BirthDate_Day、BirthDate_Month 和 BirthDate_Year。

注意：

这可能是一个不太会发生的场景，但如果刚好在同一页面有两个用于相同模型的编辑器，那么这种情况就会有 ID 冲突了，如果不进行处理，模型帮助器就可能无法解决这个问题。如果处于这种情况，可以考虑我们在第 3 章中讨论的技巧，将自定义类型的集合绑定到一个控制器方法。

如何命名这些模板？如果指定了 UIHint 特性，则由它的值确定模板名称。如果没有指定，那么 DataType 特性会优先于实际的成员类型名称。对于以下类定义，预期的模板名称是日期。如果没有视图引擎可以提供这样的视图，就会使用默认模板。图 4-7 显示了前面定义的基于表格的日期编辑器。

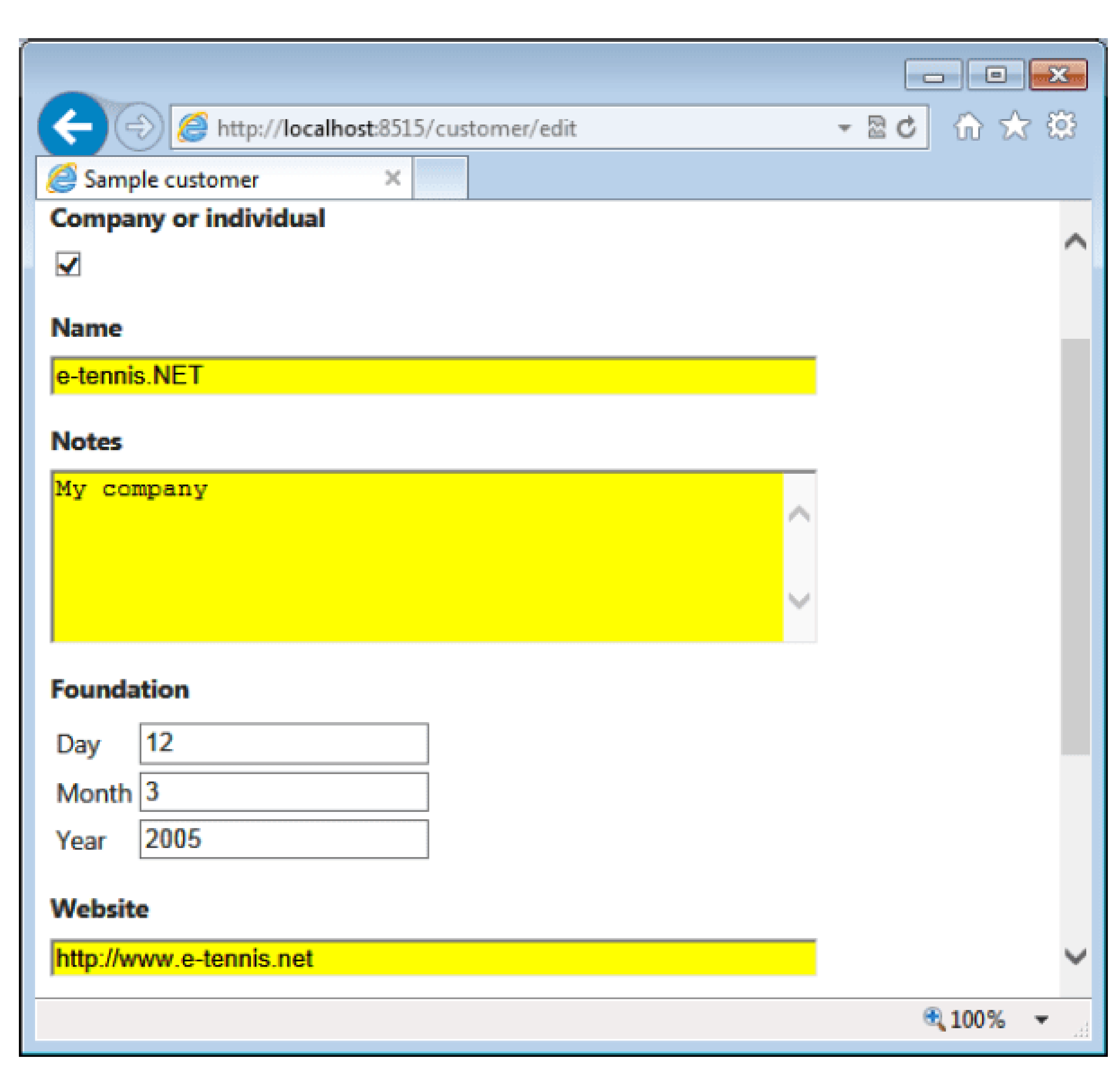


图 4-7　用于编辑日期的自定义模板

4. 只读成员

如果用 ReadOnly 特性修饰视图模型的成员，你大概会预计它不能在编辑器中编辑。你

会预计显示模板是在编辑器中使用以用于模型的。但你会惊讶地发现情况并非如此。ReadOnly 特性会被元数据提供程序正确识别，相关信息存储在模型可用的元数据中。不过出于某些原因，这并没有转换成模板提示。

可能这听起来有些奇怪，你有数据批注来指示某个指定成员是只读的，但这并不会被默认模板反映出来。有几个办法可以解决此问题。首要的是，可使用 UIHint 批注指定一个名为 readonly.cshtml 的只读模板，如下所示：

```
<span>@Model</span>
```

你需要将 readonly.cshtml 模板放在编辑器文件夹中。

这种解决方法虽然有效，但它绕过了 ReadOnly 特性的使用。这倒不是什么大问题，但你很可能想知道是否有一种解决方法可以强制元数据提供程序以不同方式处理 ReadOnly 特性。你需要一个不同的元数据提供程序，比如下面这个：

```
public class ExtendedAnnotationsMetadataProvider :
    DataAnnotationsModelMetadataProvider
{
    protected override ModelMetadata CreateMetadata(IEnumerable<Attribute>
       attributes,
       Type containerType,
       Func<Object> modelAccessor,
       Type modelType,
       String propertyName)
    {
       var metadata = base.CreateMetadata(
          attributes,
          containerType,
          modelAccessor,
          modelType,
          propertyName);

       if (metadata.IsReadOnly)
          metadata.TemplateHint = "readonly";   // Template name is arbitrary
       return metadata;
    }
}
```

你要创建一个新的类，它继承 DataAnnotationsModelMetadataProvider 类，并重载 CreateMetadata 方法。重载很简单——先调用基本方法，然后检查 IsReadOnly 属性返回的内容。如果该成员声明了是只读的，那么可以用编程方式将 TemplateHint 属性设置到你的自定义只读模板(之后确保该模板在 Views 文件夹中可用就是你的责任了)。

你必须向系统注册一个自定义的元数据提供程序。可以使用两种方式。可以把新提供程序的一个实例存储在ModelMetadataProviders类的Current属性中，如下所示(这需要在global.asax文件的Application_Start中完成)。

```
ModelMetadataProviders.Current = new ExtendedAnnotationsMetadataProvider();
```

在 ASP.NET MVC 中，你有另一个同样有效的选择：使用依赖解析器。我会在第 7 章中讨论依赖解析器的话题。现在，只需要说明 ASP.NET MVC 早期版本的所有可插拔式内部组件以及很多新的组件，都在 ASP.NET MVC 中，可以通过使用类似服务定位器的集中式服务(依赖解析器)由系统发现。服务定位器是一个通用组件，它获取一种类型(比如一个接口)，并返回另一种类型(比如实现该接口的一个具体类型)。

在 ASP.NET MVC 中，ModelMetadataProvider 类型可通过依赖解析器发现。所以，你要做的就是用量身定做的依赖项解析器注册你的自定义提供程序。依赖解析器是知道如何获得请求类型对象的一种类型。下面的代码显示了一个为此方案量身定做的简单依赖解析器：

```
public class SampleDependencyResolver : IDependencyResolver
{
    public object GetService(Type serviceType)
    {
        try
        {
            return serviceType == typeof(ModelMetadataProvider)
                ? new ExtendedAnnotationsMetadataProvider()
                : Activator.CreateInstance(serviceType);
        }
        catch
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return Enumerable.Empty<Object>();
    }
}
```

你要像这样在 global.asax 中注册解析器：

```
DependencyResolver.SetResolver(new SampleDependencyResolver());
```

请注意 ASP.NET MVC 会试图找到一个有效的 ModelMetadataProvider 实现，首先通过探

寻已注册的解析器，然后探寻 ModelMetadataProviders.Current 属性。ModelMetadataProvider 类型的双重注册是不允许的，如果出现这种情况，将会引发异常。

注意：

支持编辑器中只读属性的另一种方法是修改所使用的默认模板。很快会讨论这方面的内容。

4.2.2　用于模型数据类型的自定义模板

到目前为止，我们已经公开了作用于值的简单类型的有关显示和编辑器模板的大量细节。然而，像 EditorForModel 和 DisplayForModel 这些帮助器的内部结构却提供了一些通过模型类型的公共接口来反映和构建层级视图的固有能力。这种行为是硬编码的，但它过于简单，在实际环境中不太有用。在这一节中，我将向你展示如何重写显示和编辑器模板，使泛型看起来更像表格。换句话说，不是得到一个标签和值<div>的垂直堆叠面板，而是一个两列的表格，标签在左边，值在右边。重写的模板叫做对象。

1. 表格式模板

编写一个基于表格的布局是一个相对简单的任务，它包括模型对象属性的循环遍历。下面是 Razor 代码：

```
@inherits System.Web.Mvc.WebViewPage
@if (Model == null)
    <span>@ViewData.ModelMetadata.NullDisplayText</span>
else
{
    <table cellpadding="0" cellspacing="0" class="display-table">
    @foreach (var prop in ViewData
                    .ModelMetadata
                    .Properties
                    .Where(pm => pm.ShowForDisplay
                    && !ViewData.TemplateInfo.Visited(pm)))
    {
        <tr>
            <td>
                <div class="display-label">
                    @prop.GetDisplayName()
                </div>
            </td>
            <td>
                <div class="display-field">
                    <span>@Html.Display(prop.PropertyName)</span>
```

```
                </div>
            </td>
        </tr>
    }
    </table>
}
```

如果模型为 null，你只需要为模型类发出默认的 null 文本。如果不为 null，则要循环处理所有属性，并为所有没有被预先访问且设置用于显示的属性创建一个表格行。对于每一个属性，你再以递归方式调用显示 HTML 帮助器。可以任意设计每一小块标记的样式。在此示例中，我会介绍一个新的表格级别的样式，称为显示表格。图 4-8 显示了其结果。

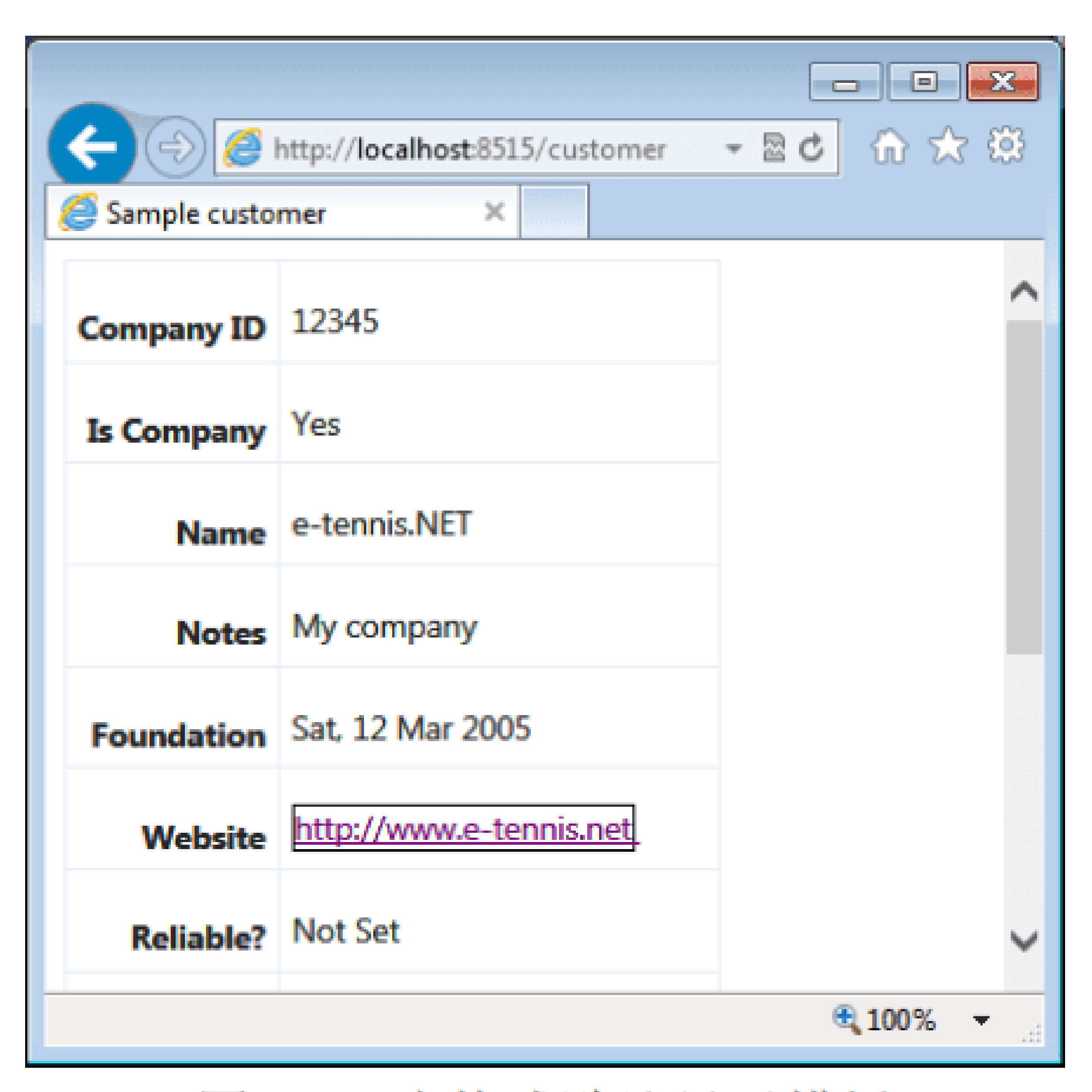


图 4-8　表格式默认显示模板

编辑器模板更复杂一些，因为它要负责验证错误和必需的字段。该模板有额外的一列用于指示哪些字段是必需的(基于Required批注)，且每个编辑器的顶部都有一个验证消息的标签。下面是源代码：

```
@inherits System.Web.Mvc.WebViewPage
@if (Model == null)
    <span>@ViewData.ModelMetadata.NullDisplayText</span>
else {
    <table cellpadding="0" cellspacing="0" class="editor-table">
    @foreach (var prop in ViewData
                    .ModelMetadata
                    .Properties
                    .Where(pm => pm.ShowForDisplay
                    && !ViewData.TemplateInfo.Visited(pm)))
    {
        <tr>
```

```
                <td>
                    <div class="editor-label">
                        @prop.GetDisplayName()
                    </div>
                </td>
                <td width="10px"> @(prop.IsRequired ? "*" : "") </td>
                <td>
                    <div class="editor-field">
                        @if(prop.IsReadOnly)
                        {
                            <span>@Html.Display(prop.PropertyName)</span>
                        }
                            else
                            {
                                <span>@Html.Editor(prop.PropertyName)</span>
                                <span>@Html.ValidationMessage(prop.PropertyName,
                                 "*")</span>
                            }
                    </div>
                </td>
            </tr>
        }
    </table>
}
```

图 4-9 显示了实际运行中的编辑器。

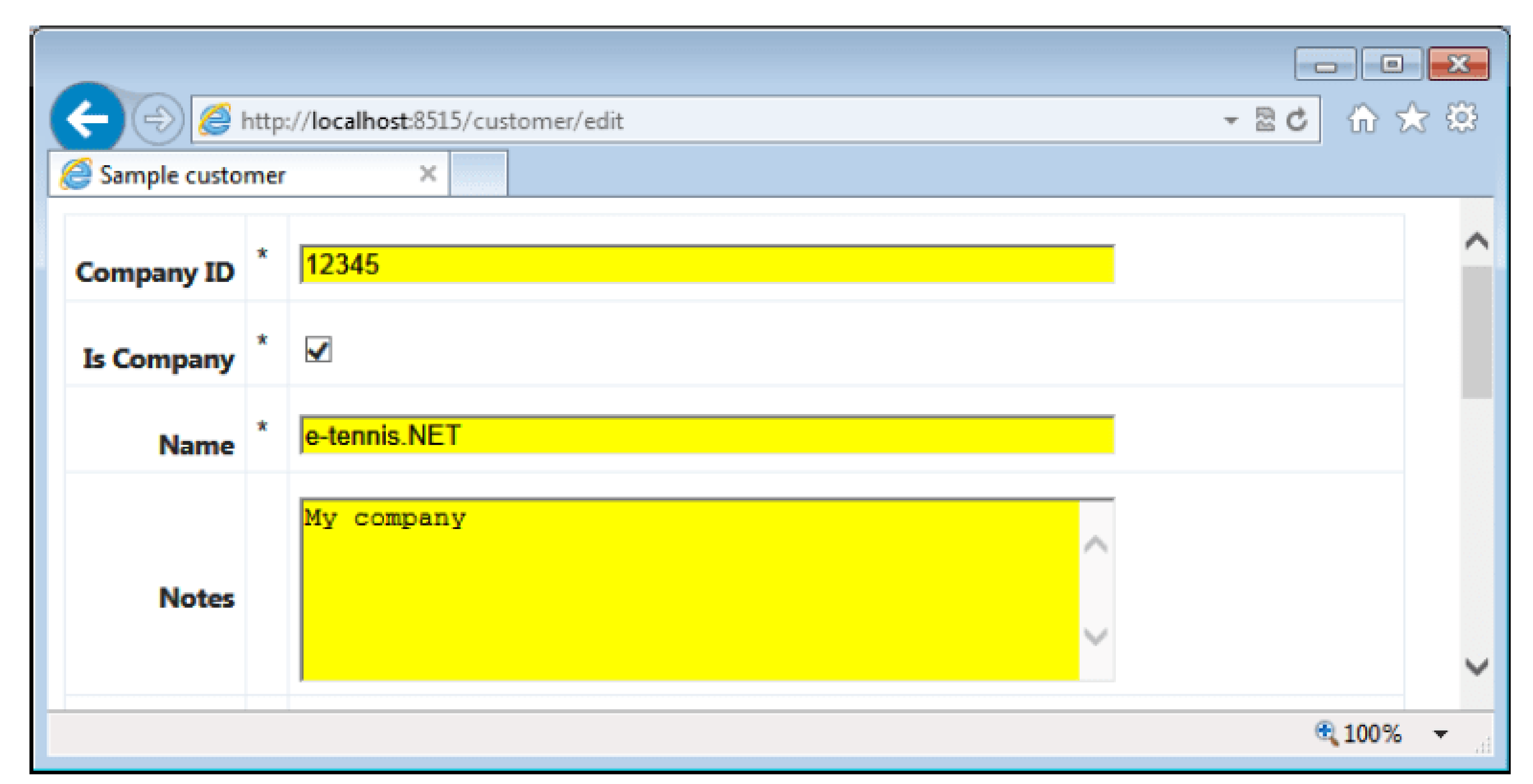


图 4-9　表格式默认编辑器模板

2. 处理嵌套模型

最后要说明的一点是，当模型类型具有嵌套模型时所预期的模板行为。到目前为止以递

归方式显示的代码找出了嵌套模型的属性，并将它们像通常那样呈现。设想一下你有下面的模型。

```
public class CustomerViewModel
{
   ...
   public ContactInfo Contact {get; set;}
}

public class ContactInfo
{
   public String FullName {get; set;}
   public String PhoneNumber {get; set;}
   public String Email {get; set;}
}
```

图 4-10 显示了其结果。

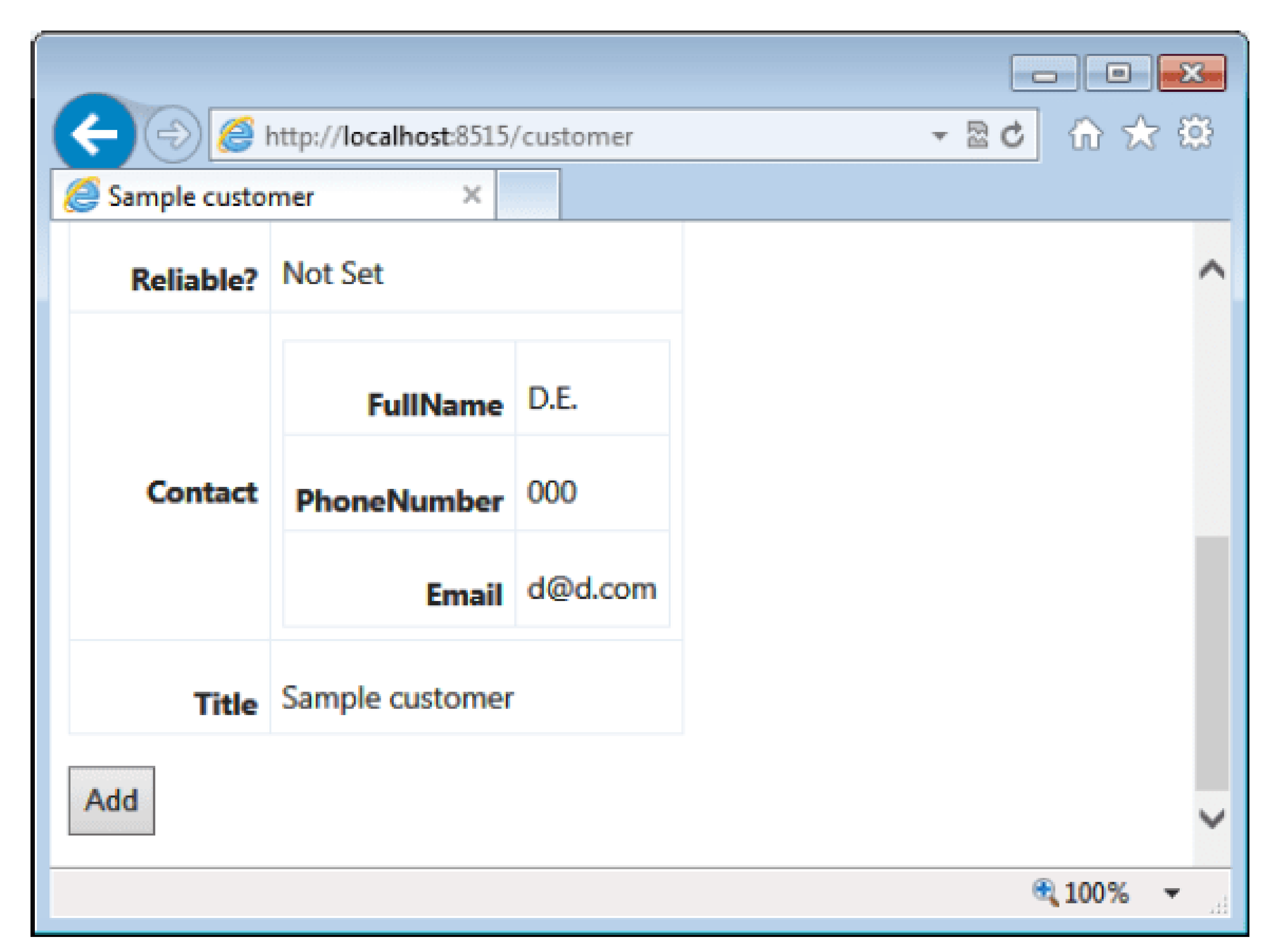


图 4-10 嵌套模型

作为开发人员，可以在某种程度上控制这一行为。通过在模板的代码中添加一个额外的 if 分支，可将嵌套级别限制在你希望的级别(此示例中是 1)。

```
@if (Model == null)
    <span>@ViewData.ModelMetadata.NullDisplayText</span>
else
if (ViewData.TemplateInfo.TemplateDepth > 1)
{
    <span>@ViewData.ModelMetadata.SimpleDisplayText</span>
}
else
```

```
{ ... }
```

TemplateDepth 属性会衡量允许的嵌套级别。随着该代码的启用，如果嵌套级别高于 1，则要采用子模型的简化表示。可以通过重写子模型类型的 ToString 来控制这种表示。

```
public class ContactInfo
{
    ...
    public override string ToString()
    {
        return "[ContactInfo description here]";
    }
}
```

或者，可以选择一列并将之用于呈现。通过使用子模型类型上的 DisplayColumn 特性来选择该列。

```
 [DisplayColumn("FullName")]
public class ContactInfo
{
   ...
}
```

DisplayColumn 特性优先级高于 ToString 的重载。

注意：

如你所见，HiddenInput 特性可以隐藏一个来自视图的指定成员，同时通过一个隐藏字段上传其内容。如果希望帮助器忽略指定的成员，则可以使用 ScaffoldColumn 特性并给它传递一个 false 值。

4.3　输入验证

编程的书籍往往充斥着诸如“输入是不可信的”语句，当然，专门介绍输入表单的一章不可能漏掉有关输入验证的部分。如果想要讨论它的必要性，就应该专注于服务器端验证，想想可用的最好技术是什么，以最有效的方式帮助应用程序和业务域的验证。

然而，话虽如此，但确实没有理由跳过客户端验证步骤，它恰恰能够阻止明显无效的数据进入服务器且耗费宝贵的 CPU 周期。因此，你应该对客户端和服务器端的验证都有兴趣。但是，即使在一般复杂的 Web 应用程序中，验证也至少会应用于两个层面：验证从浏览器接收的输入，以及验证系统后端要存储的数据。

这两个层面有时候可能是同一个，但这并不是现实中以及一些很酷的技术书籍和指南之

外常有的情况。除非域模型本质上一对一地与存储一致，且用户界面大多处理的是 Create/Read/Update/Delete(CRUD)，否则你必须考虑(并制定计划)两个层面的验证：呈现和业务。

在本章的剩余部分，我将回顾其他一些在 ASP.NET MVC 系统中良好集成的数据批注特性。

4.3.1　使用数据批注

如前所述，数据批注是一组特性，可以使用它们批注任何.NET 类的公共属性，以任意一个相关客户端代码可以读取和使用的方式。特性可以分为两个主要类别：显示和验证。我们已经讨论了显示特性配合元数据提供程序在 ASP.NET MVC 中所发挥的作用。但是，在深入探究验证特性之前，让我们先多了解一些有关 ASP.NET MVC 中数据验证的知识。

1. 验证提供程序的基础结构

第 3 章阐述了控制器如何通过模型绑定子系统接收它们的输入数据。模型绑定器会把请求数据映射到模型类，与此同时，它会根据模型类上设置的验证特性集对输入值进行验证。

验证是通过提供程序实现的。默认的验证提供程序基于数据批注，是 DataAnnotationsModelValidatorProvider 类。我们看看可以使用哪些默认验证提供程序能够理解的特性。

表 4-2 列出了模型类上表示验证条件的最常用数据批注特性。

表 4-2　用于验证的数据批注特性

特　性	描　述
Compare	检查模型中的两个指定属性是否具有相同值
CustomValidation	根据指定的自定义函数检查值
EnumDataType	检查指定枚举类型中的任意一个值是否可以匹配某值
Range	检查某值是否落在指定区间内。默认是数值型，不过也可以配置它以处理日期范围
RegularExpression	检查某值是否匹配指定表达式
Remote	执行一个到服务器的 Ajax 调用并检查某值是否可以接收
Required	检查是否为属性分配了一个非空值。可以对其进行配置以便分配了空值时报错
StringLength	检查字符串是否超出了指定长度

所有这些特性均从同一个基类派生而来，那就是 ValidationAttribute。你很快会发现，也可以使用此基类来创建自定义验证特性。

可以使用这些特性来修饰输入表单中正在使用的类成员。对于整个运行机制，你需要有在复杂数据类型中接收数据的控制器方法，即如下所示的 Memo 控制器：

```
public ActionResult Edit()
```

```
{
    var memo = new Memo();
    return View(memo);
}

[HttpPost]
public ActionResult Edit(Memo memo)
{
    // ModelState dictionary contains error messages
    // for any invalid value detected according to the annotations
    // you might have in the Memo model class.

    return View(memo);
}
```

模型绑定器对象会在将传递值绑定到Memo模型类的同时编辑ModelState字典。对于正在映射到Memo类实例的任何无效的传递值，绑定器会自动在 ModelState字典中创建一个条目。传递值是否有效取决于当前已注册验证提供程序所返回的结果。默认的验证提供程序会基于你在Memo模型类上设置的批注来进行响应。最后，如果下一个视图要使用ValidationMessage帮助器，那么错误消息会自动显示。这正是你使用EditorForModel创建输入表单的情况。

2. 修饰模型类

下面的代码显示了一个示例类——前面提到过的 MemoDocument 类——它将数据批注广泛用于验证：

```
public class MemoDocument : ViewModelBase
{
    public MemoDocument()
    {
        Created = DateTime.Now;
        Category = Categories.Work;
    }

    [Required]
    [StringLength(100)]
    public String Text { get; set; }

    [Required]
    [Range(1, 5]
    public Int32 Priority { get; set; }

    [Required]
```

```
    public DateTime Created { get; set; }

    [EnumDataType(typeof(Categories))]
    [Required]
    public Categories Category { get; set; }

    [StringLength(50, MinimumLength=4)]
    [RegularExpression(@"\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b")]
    public String RelatedEmail { get; set; }
}

public enum Categories
{
    Work,
    Personal,
    Social
}
```

对于这个类，如果 Text 属性超过 100 个字符或者为空，就会收到错误消息。同样，Priority 成员必须是介于 1 和 5(包含 1 和 5)之间的整数，且不能省略 Created 日期。RelatedEmail 成员可以为空；但是，如果指定任何文本，则该文本必须介于 4~50 个字符长度之间并匹配正则表达式。最后，Category 成员必须包含一个字符串，其会对 Categories 枚举类型中的一个常量进行估算。图 4-11 显示了一个示例 Memo 的验证。

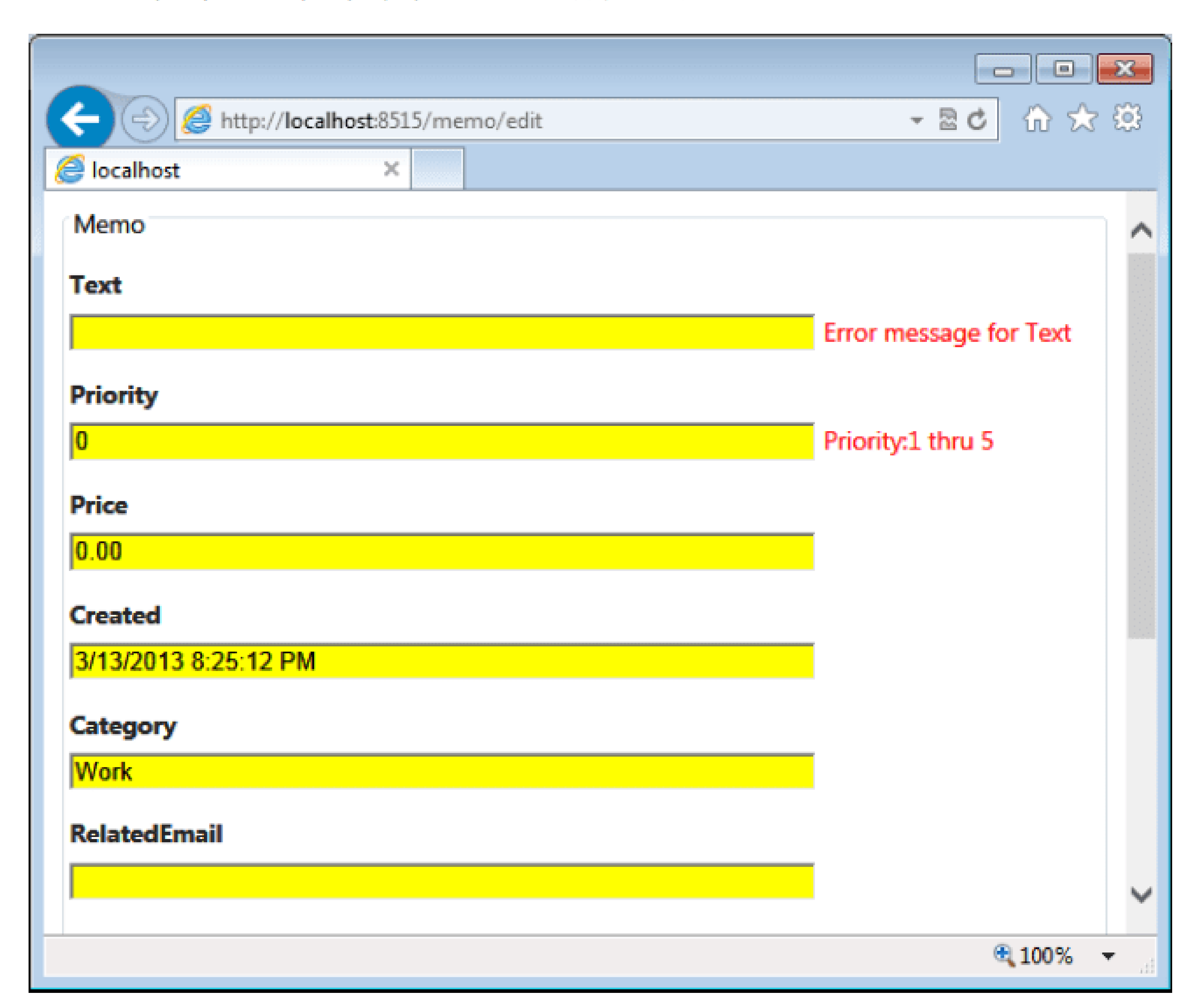


图 4-11 验证消息

3. 处理枚举类型

你可能好奇为什么 Category 字段要作为普通字符串来编辑。如果能提供一个下拉列表，它会变得更加智能。但这一效果费一番周折才能得到。EnumDataType 由验证提供程序识别，它会确保值是属于枚举的；它会被编辑器忽略。如果想要枚举值的下拉列表，则需要编写一个 String.cshtml 的自定义模板并将其放入 EditorTemplates 文件夹中。因为枚举类型要与字符串配对，因此你要重写 String.cshtml(或.aspx)以更改枚举编辑的方式。在代码中，你要根据 Model 属性的实际类型来确定标记。下面是你需要的简单代码：

```
@model Object
@if (Model is Enum)
{
   <div class="editor-field">
      @Html.DropDownList("", new SelectList(Enum.GetValues(Model.GetType())))
      @Html.ValidationMessage("")
   </div>
}
else
{
    @Html.TextBox("", ViewData.TemplateInfo.FormattedModelValue,
          new { @class = "text-box single-line" })
}
```

图 4-12 显示了使用刚创建的编辑器模板时的同一个表单。

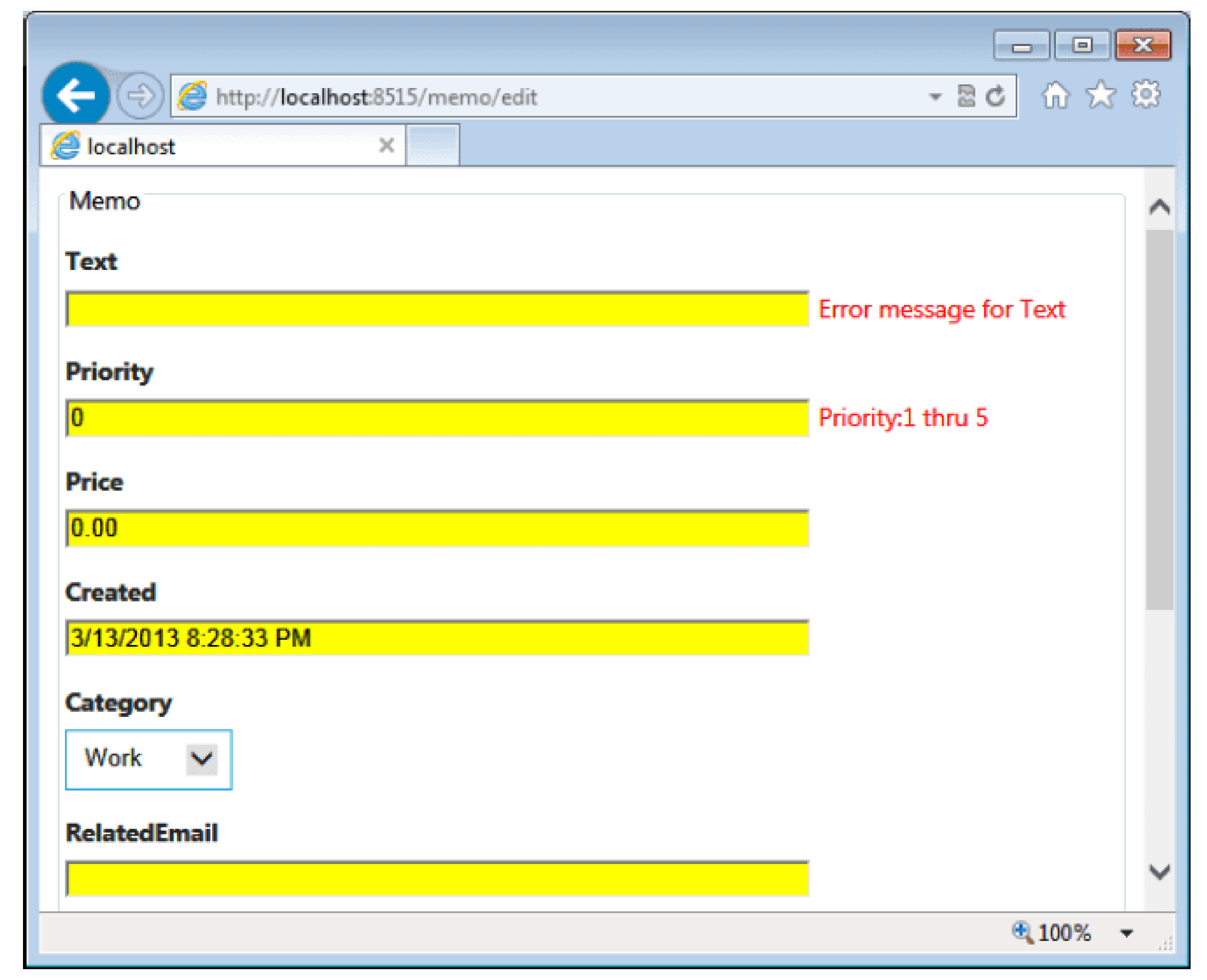


图 4-12　混合验证与显示数据批注

4. 控制错误消息

在图 4-12 中，可以看到一些说明哪里出错的消息。这些消息可以由你进行全面控制。每个特性都有一个 ErrorMessage 属性可用来设置文本。但是请注意，特性属性只能接受常量字符串。

```
[Required(ErrorMessage = "A description is required.")]
public String Text { get; set; }
```

你可能不喜欢在纯字符串中穿插类定义的主意。从错误消息中解耦类定义的一种方法是使用资源。使用资源可使你在不接触类的情况下对文本进行脱机更改。它还有助于区域设置，但它不会提供对正在显示的文本的编程控制。要进行编程控制，你唯一的选择是在控制器方法中编辑 ModelState 字典，如下所示：

```
[HttpPost]
public ActionResult Edit(MemoDocument memo)
{
    if (!ModelState.IsValid)
    {
        var newModelState = new ModelStateDictionary();

        // Create a new model-state dictionary with values you want to overwrite
        newModelState.AddModelError("Text", "...");
        newModelState.AddModelError("Priority", "...");
        ...

        // Merge your changes with the existing ModelState
        ModelState.Merge(newModelState);
    }
    return View(memo);
}
```

基本上，你要创建一个新的模型状态字典，然后在模型绑定阶段将其与计算过的字典合并。在合并字典时，新字典中的值会复制到 ModelState 字典中，并覆盖所有的现有值。

如果有持续的消息为你所用，那么你需要做的就是避免软件中出现魔幻字符串，可以使用资源文件，顺便说一下，资源文件还设置了状态以便易于区域化。每个验证特性都会接受一个表示为资源索引的错误消息：

```
[Required(ErrorMessageResourceName="MustSetPriority",
      ErrorMessageResourceType = typeof(Strings))]
[Range(1, 5, ErrorMessageResourceName="InvalidPriority",
      ErrorMessageResourceType=typeof(Strings))]
public Int32 Priority { get; set; }
```

可通过两个参数来表示资源：资源容器数据类型和资源的名称。前者通过 ErrorMessageResourceType 属性来表示；后者由 ErrorMessageResourceName 属性表示。当在 ASP.NET MVC 项目中添加一个资源文件时，微软 Visual Studio 设计器就会创建一个容器类型，将字符串公开为公共成员。这就是要分配给 ErrorMessageResourceType 属性的类型。默认情况下，自动生成类型的名称与资源(.resx)文件的名称是匹配的。

4.3.2　高级数据批注

数据批注的优点是在你定义了模型类型的特性后，差不多就完事了。后面的任务几乎都自动完成，多亏 ASP.NET MVC 基础设施具有对数据批注的深刻理解。然而在这个不完美的世界，你真正需要的东西都不是那么容易就能得到的。数据批注也一样，它对不少相对简单的情况能处理得很好，但在很多真实的应用程序中却给你留下了需要你自己去处理的额外工作。接下来我们探讨几个可构建在数据批注之上的高级功能。

1. 跨属性验证

到目前为止我们所理解的数据批注是用于验证单个字段内容的特性。这当然是有用的，但对于现实情况下属性值存储在另一个属性中的内容验证，它却起不了什么作用。跨属性验证需要一些特定于上下文的代码。问题是：如何编写代码以及在哪里编写？

能即刻想到的解决方案大致是下面这样的：

```
public ActionResult Edit(MemoDocument memo)
{
    if (ModelState.IsValid)
    {
        // If here, then properties have been individually validated.
        // Proceed with cross-property validation, and merge model-state
           dictionary
        //
        // to reflect feedback to the UI.
        ...
    }
}
```

如果从绑定器接收到的模型状态是有效的，那么所有的修饰属性就都通过了验证阶段。因此，孤立地处理每个属性是可以的。你要继续进行特定于上下文的、跨属性的验证，将错误添加到新的状态字典，并将其合并到现有的 ModelState 中。

你可能乐于知道 CustomValidation 特性正是服务于这样的目的，总体而言，可以将它看成我前面推荐方法的快捷方式。思考下面的代码：

```
[Required]
[EnumDataType(typeof(Categories))]
[CustomValidation(typeof(MemoDocument), "ValidateCategoryAndPriority")]
public Categories Category { get; set; }
```

CustomValidation 特性有两个参数：分别是类型和方法名称。类型也可以是你正在修饰的同一个模型。方法必须是带有以下签名的公共静态方法：

```
public static ValidationResult MethodName(Categories category)
public static ValidationResult MethodName(Categories category,
   ValidationContext context)
```

使用第一个重载等同于定义用于单个值的自定义验证逻辑。某种意义上相当于创建了一个自定义数据批注特性——这有助于更快地编写，但在签名方面比较严格。更有趣的是第二个重载。事实上，通过 ValidationContext 参数，可以获得对模型对象的引用，并可以任意检查多个属性。

```
public static ValidationResult ValidateCategoryAndPriority(
                Categories category, ValidationContext context)
{
    // Grab the model instance
    var memo = context.ObjectInstance as MemoDocument;
    if (memo == null)
       throw new NullReferenceException();

    // Cross-property validation
    if (memo.Category == Categories.Personal && memo.Priority > 3)
       return new ValidationResult("Category and priority are not
        consistent.");
    return ValidationResult.Success;
}
```

可将 CustomValidation 特性附加到一个单独的属性，也可以附加到类。在前面的示例中，该特性是附加到 Category 的，但它要确保如果值是 Personal，则 Priority 属性必须设置成不大于 3 的值。如果验证失败，那么模型状态字典中的条目会把 Category 用作键值。如果把 CustomValidation 附加到类，则要注意，只有当所有的单个属性都不出问题，验证才会执行。这正是前面所介绍的模式的声明性对应：

```
 [CustomValidation(typeof(MemoDocument), "ValidateMemo")]
public class MemoDocument
{
   ...
}
```

下面是 CustomValidation 特性附加到类时所对应的方法签名：

```
public static ValidationResult ValidateMemo(MemoDocument memo)
{
  ...
}
```

当你在类级别使用 CustomValidation 时，会遇到捕获错误消息的问题，因为错误消息通常与属性相关联。可以使用 Html.ValidationSummary 帮助器，它会将所有的错误消息提取出来，无论是否为原始属性，从而轻松地解决掉这个问题。本章稍后将讨论类级别验证的主题。

最后，Compare 特性可用来更快速地服务于具体的跨属性方案；比如，当你需要确保一个属性的值与另一个属性的值相匹配时。典型的例子是重新键入一个新密码：

```
[Required]
[DataType(DataType.Password)]
public String NewPassword {get; set;}

[Required]
[DataType(DataType.Password)]
[Compare("NewPassword")]
public String RetypePassword {get; set;}
```

这一比较是通过在特定类型上使用 Equals 方法来实现的。

2. 创建自定义验证特性

CustomValidation 特性会强制你在不增加任何额外参数的前提下验证存储在属性中的值。问题是没有足够多的对模型对象上其他属性的访问权——而这个问题在使用 ValidationContext 是不会遇到的——却丰富了那些定义了附加特性级别参数的特性的签名。例如，假设你要验证一个数字以确保它是偶数。另外根据需要，你还要打开这个特性检查它是否也是 4 的倍数。这时你需要一个如下所示的特性：

```
[EvenNumber(MultipleOf4=true)]
public Int32 MagicNumber {get; set;}
```

没有办法在由 CustomValidation 所识别的签名中传递一个可选的 Boolean 值。最终要说的是，CustomValidation 特性与自定义验证特性之间的区别是后者的目的在于易于重复使用。下面显示了如何编写一个自定义数据批注特性：

```
 [AttributeUsage(AttributeTargets.Property)]
public class EvenNumberAttribute : ValidationAttribute
{
```

```
    // Whether the number has to be checked also for being a multiple of 4
    public Boolean MultipleOf4 { get; set; }

    public override Boolean IsValid(Object value)
    {
        if (value == null)
            return false;

        var x = -1;
        try
        {
            x = (Int32) value;
        }
        catch
        {
            return false;
        }

        if (x % 2 > 0)
            return false;
        if (!MultipleOf4)
            return true;

        // Is multiple of 4?
        return (x % 4 == 0);
    }
}
```

你要创建一个从 ValidationAttribute 继承的类，并重写 IsValid 方法。如果需要额外的参数，比如 MultipleOf4，就定义公共属性吧。

在 ASP.NET MVC 中，可以创建执行跨属性验证的自定义特性。你只需要重写 IsValid 方法的重载，使其稍加改变：

```
protected override ValidationResult IsValid(Object value, ValidationContext
context)
```

使用 ValidationContext 对象上的属性，可以获得对整个模型对象的访问，从而执行充分的验证。

3. 启用客户端验证

所有示例处理的都是贯穿一个回传或 Ajax 表单的任务。要将其与 Ajax 一起使用，你只需要对显示编辑器的视图做一点小的更改，以便它们使用 Ajax.BeginForm 而非

Html.BeginForm。另外，控制器方法应该返回一个部分视图而不是完整视图，如下所示：

```
 [HttpPost]
public ActionResult Edit(MemoDocument memo)
{
    if (Request.IsAjaxRequest())
        return PartialView("edit_ajax", memo);
    return View(memo);
}
```

下面的代码显示了如何将原始视图转换成 edit_ajax 部分视图：

```
@model DataAnnotations.ViewModels.Memo.MemoDocument
@{
    Layout = "";  // Drop the master page
}

@using (Ajax.BeginForm("edit", "memo",
     new AjaxOptions() { UpdateTargetId="memoEditor" }))
{
    <div id="memoEditor">
    <fieldset>
        <legend>Memo</legend>
        @Html.EditorForModel()
        <p>
        <input type="submit" value="Create" />
        </p>
    </fieldset>
    </div>
}
```

以这种方式，你的表单就会在服务器上进行验证，但不会刷新整个页面。然而，至少对于某些基本批注来说，这有助于开启客户端验证，以便在数据是明显无效时(例如，所需的字段为空)不会启动 HTTP 请求。

要开启客户端验证，需要执行几个简单的步骤。首先，确保你的 web.config 文件包含以下内容：

```
<appSettings>
   <add key="ClientValidationEnabled" value="true" />
   <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

注意：

这两个设置都是默认设置。此外，你需要链接两个 JavaScript 文件：jquery.validate.js(jQuery 验证插件)和 jquery.validate.unobtrusive.js，或者它们的压缩版。查找和添加这些文件的最简单方法是 NuGet。不用说，当务之急就是链接 jQuery 库。

重要提示：

当你开启客户端验证时，所有内置的数据批注特性都会获得一个客户端行为，并尽可能在浏览器中执行其在 JavaScript 中的验证。如果验证失败，就不会向 Web 服务器发出请求。然而，像 EvenNumber 一样的自定义验证特性不会自动照此原理工作。要添加用于自定义特性的客户端验证，你还需要实现一个额外的接口，即 IClientValidatable，这会在后面讲到。

4. 基于文化的客户端验证

在面对数据类型验证问题时，你可能会遇到额外的全局化问题。例如，请思考下面的问题：

```
public Decimal Price {get; set;}
```

编辑器会正确处理类型，并显示“0.00”的默认格式。然而，如果输入“0,8”，将逗号作为小数点分隔符，那么你的输入将被拒绝，表单不会提交。可以看到，设置客户端验证上的正确区域性是一个问题。jQuery Validation 插件在客户端上默认为美国区域设置；相反，在服务器端，它取决于线程上 Culture 属性的值(参见第 5 章“ASP.NET MVC 应用程序的特性”有关区域设置和全局设置的详细信息)。

要支持客户端上的某指定区域，必须首先链接官方 jQuery 全局设置插件，以及你想要的指定区域的脚本文件。在常规验证脚本之后必须包含这两个文件。此外，必须指示全局设置插件你要使用哪个区域。最后，必须告知验证程序插件，当它解析数字时需要使用全局设置信息。

```
<script type="text/javascript">
    $.validator.methods.number = function (value, element) {
        if (Globalization.parseFloat(value)) {
            return true;
        }
        return false;
    }
    $(document).ready(function () {
        $.culture = jQuery.cultures['it-IT'];
        $.preferCulture($.culture.name);
        Globalization.preferCulture($.culture.name);
    });
</script>
```

有了此代码后，就可以使用指定区域设置来输入小数。进一步使用此方法，就可以在区域基础上自定义大部分的客户端验证工作。

5. 远程验证属性

验证可能发生在客户端上，但你应该把它看作为页面避免一些高强度 HTTP 请求的一种方式。为安全起见，你应该始终验证服务器上的所有数据。然而，要给用户带来更好的体验，你会希望在不离开浏览器的情况下执行服务器端验证。典型的例子是当用户注册某个服务并输入昵称之时。该名称必须是唯一的，而其唯一性只能在服务器上进行验证。如果可以实时告知用户该昵称是否已被使用，是不是很酷呢？这样一来，用户便能及时更改，并且避免在提交注册请求时出现恼人的意外。

数据批注提供了一个有助于此功能编码的特性：Remote 特性。附加到一个属性，该特性就能调用某控制器上的方法并预期一个 Boolean 响应。该控制器方法会接收要验证的值以及一个额外的相关字段列表。下面是一个示例：

```
[Remote("CheckCustomer", "Memo",
        AdditionalFields="Country",
        ErrorMessage="Not an existing customer")]
public String RelatedCustomer { get; set; }
```

当在客户端上验证 RelatedCustomer 时，代码会隐式地把 jQuery 调用放置到 Memo 控制器上的 CheckCustomer 方法。如果响应是否定的，则显示指定的错误消息。

```
public ActionResult CheckCustomer(String relatedCustomer)
{
   if (CustomerRepository.Exists(relatedCustomer))
      return Json(true, JsonRequestBehavior.AllowGet);
   return Json(false, JsonRequestBehavior.AllowGet);
}
```

该控制器必须返回封装在一个 JSON 有效载荷内的 true/false。如果已经指定了附加字段，则它们会被添加到 URL 的查询字符串并遵循 ASP.NET MVC 的经典模型绑定规则。多个字段要用逗号隔开。下面是一个 URL 示例：

```
http://yourserver/memo/checkcustomer?relatedCustomer=dino&Country=...
```

每一次输入字段在被修改后失去焦点的时候，就会触发 Ajax 调用。以 Remote 特性修饰的属性会接受客户端验证约定，并且在输入一个有效值之前不允许表单回传。

重要提示：

数据批注在编译时只能静态指定。目前，尚没有办法从外部数据源读取特性并将它们绑

定到运行中的模型属性。要开启此功能，你可能需要考虑用一种可以从不同源中读取元数据进行验证的自定义验证提供程序，替换掉默认的验证提供程序。

我发现相较于表示层，这更像是业务层验证的现实问题。在业务层验证中，随着需求的变化，你可能需要注入业务规则，同时也需要更大的灵活性。在业务层验证中，我经常使用企业库验证应用程序块(Enterprise Library Validation Application Block)，它可以从配置文件中读取验证特性(稍后会回头来探讨企业库)。

4.3.3　自我验证

数据批注会试图将正在从表单传递的数据验证过程自动化。大多数时候，你只会使用常备特性，并毫不费力地获得错误消息。在其他情况下，你需要以稍微高一点的开发成本创建自己的特性，但这样做的同时也创建了非常适应现有基础设施的组件。此外，数据批注的目的在于大多数时候在属性级别工作。在 ASP.NET MVC 中，你始终可以通过验证上下文来访问整个模型；然而，最后当验证变得复杂时，许多开发人员宁愿选择一个手工定制的验证层。

换句话说，你要停止众多数据批注组合中的分隔验证，并将所有处理任务移到一个单独的地方——一种从控制器内的服务器上调用的方法：

```
public ActionResult Edit(MemoDocument memo)
{
   if (!Validate(memo))
     ModelState.AddModelError(...);
   ...
}
```

MemoDocument 类可能有也可能没有属性批注。以我看来，如果选择自我验证，为明确起见你应该停止使用数据批注。不过在任何情况下，自我验证都不会阻止你使用数据批注。

1. IValidatableObject 接口

在面对构建自我验证层的任务时，可以发挥你的创造力，并选择最适合你的应用程序模型。ASP.NET MVC 会试图推荐一种方法。基本上，ASP.NET MVC 会保证任何实现 IValidatableObject 接口的模型类都会自动验证，无须开发人员显式调用验证。该接口如下所示：

```
public interface IValidatableObject
{
  IEnumerable<ValidationResult> Validate(ValidationContext
    validationContext);
}
```

如果检测到了接口，验证提供程序会在模型绑定步骤期间调用 Validate 方法。

ValidationContext类型的参数使整个模型可用于任何类别的跨属性验证。

重要提示：

如果模型也被数据批注修饰，其中某些属性没有处于有效状态，则不会调用 Validate 方法(为了避免错误，如果选择了 IValidatableObject，我建议你完全放弃数据批注)。

使用 IValidatableObject 在功能上等同于在类级别上使用 CustomValidation 特性。唯一的区别是有了 IValidatableObject，就可以在使用你自己的架构并维持数据批注独立性的情况下，利用单个方法实现一个验证层。

2. 集中式验证的优势

当你有非常复杂的验证逻辑，其中跨属性验证占主导地位时，将每个属性的验证和类级别的验证混合可能会引发最终用户不愉快的经历。如前所述，在所有属性都单独得到了验证前，不会启动类级别的验证。这就是说用户最初会看到几个与属性相关的错误消息，随后只要这些错误被修改，用户就会认为这些错误是无碍的。相反，他们可能会因为跨属性验证而获得一个全新的错误组。用户完全不会知道其他的可能错误，除非它们显示出来。为了避免这一点，如果类级别的验证占主导地位，就请关注类级别验证，减少对每个属性验证的关注。

要实现类级别验证，在类级别进行 IValidatableObject 和 CustomValidation 之间的选择完全取决于你。多年来，我都使用自己的接口，它看上去就和今天的 IValidatableObject 几乎一模一样。

3. IClientValidatable 接口

当客户端验证处于活动状态时，自定义验证特性不会产生任何明显的效果。换句话说，自定义特性本身不能运行任何客户端代码来验证浏览器中的值。然而并不是说你不能添加这种能力。只是它需要更多代码。下面是如何扩展自定义特性从而为客户端启用这种能力的代码：

```
AttributeUsage(AttributeTargets.Property)]
public class ClientEvenNumberAttribute : ValidationAttribute,
  IClientValidatable
{
    ...

    // IClientValidatable interface members
    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
            ModelMetadata metadata, ControllerContext context)
    {
       var errorMessage = ErrorMessage;
```

```
        if (String.IsNullOrEmpty(errorMessage))
            errorMessage = (MultipleOf4 ? MultipleOf4ErrorMessage :
              EvenErrorMessage);

        var rule = new ModelClientValidationRule
                       {
                          ValidationType = "iseven",
                          ErrorMessage = errorMessage
                       };
        rule.ValidationParameters.Add("multipleof4", MultipleOf4);
        yield return rule;
    }
}
```

你只需要让它实现一个新的接口：IClientValidatable 接口。该方法会返回一个验证规则集。每个规则都具有一条错误消息、一个用于验证的 JavaScript 函数名以及一个 JavaScript 代码的参数集的特征。参数和函数名称要小写。

接着，你要编写一些在客户端执行验证的 JavaScript 代码。最好通过使用 jQuery 和 jQuery 验证插件来编写。下面是启用先前要在客户端运行的EvenNumber特性所需的JavaScript代码：

```
$.validator.addMethod('iseven', function (value, element, params) {
    var mustBeMultipleOf4 = params.multipleof4;
    if (mustBeMultipleOf4)
        return (value % 4 === 0);
    return (value % 2 === 0);
});

$.validator.unobtrusive.adapters.add('iseven', ['multipleof4',
  'more_parameters'],
    function (options) {
        options.rules['iseven'] = options.params;
        options.messages['iseven'] = options.message;
    });
```

addMethod 函数会注册执行 iseven 规则的验证回调。验证回调会接收要验证的值和先前添加到 IClientValidatable 实现中的规则的所有参数。

另外，你需要指定适配器来生成 JavaScript 自由标记的验证方案。添加适配器需要你指明函数的名称、其参数和错误消息。

不再需要其他任何东西了；如果它不能正常运行，首先检查所需的脚本是否都可用，然后检查 JavaScript 代码中和 IClientValidatable 的 C#实现中函数和参数的名称。请注意如果出现什么问题，你也不会收到任何 JavaScript 错误。

4. 动态服务器端验证

验证就是带有条件的代码，因此最终，它就是把一些 if 语句和返回的 Boolean 值结合在一起。用纯代码编写验证层，不使用任何专用框架或技术在实践中也许可行，但在设计上未必是个好点子。这样编写的代码会很难读，也难以扩展，即使最近发布的流畅代码库能使它变得稍容易一些。

受制于实际业务规则，验证是一个极不稳定的过程，并且你的实现必须要考虑到这一点。最终，这不仅与编写验证代码有关，还与要用于针对不同规则的相同数据验证代码有关。数据批注是一个可能的选择；另一种有效选择是微软企业库中的验证应用程序块(Validation Application Block，VAB)。

数据批注和 VAB 有许多共同之处。两个框架都基于特性，且都可以使用表示自定义规则的自定义类来加以扩展。这两种情况下，你都可以定义跨属性验证。最后，两个框架都有一个验证程序 API，用于评估实例并返回错误的列表。那么，区别在哪里呢？

数据批注是.NET Framework 的一部分，并且不需要任何单独的下载。企业库是一个单独的 NuGet 包。它在大型项目中可能算不了什么，但仍有一个问题，因为在某些企业方案中可能还需要额外的申请批准。

在我看来，VAB 有一方面是优于数据批注的：它可以通过 XML 规则集充分配置。XML 规则集是配置文件中的一个条目，可以按照需要在其中描述验证。更不用说，可用声明方式进行更改甚至不用触及你的代码。下面是一个规则集的示例。

```
<validation>
  <type assemblyName="..." name="Samples.DomainModel.Customer">
    <ruleset name="IsValidForRegistration">
      <properties>
        <property name="CompanyName">
          <validator type="NotNullValidator" />
          <validator lowerBound="6" lowerBoundType="Ignore"
                     upperBound="40" upperBoundType="Inclusive"
                     messageTemplate="Company name cannot be longer ..."
                     type="StringLengthValidator" />
        </property>
        <property name="Id">
          <validator type="NotNullValidator" />
        </property>
        <property name="PhoneNumber">
          <validator negated="false"
                     type="NotNullValidator" />
          <validator lowerBound="0" lowerBoundType="Ignore"
                     upperBound="24" upperBoundType="Inclusive"
                     negated="false"
```

```
                    type="StringLengthValidator" />
            </property>
            ...
          </properties>
        </ruleset>
      </type>
    </validation>
```

规则集列出了你希望应用于指定类型上的指定属性的特性。要在代码中验证规则集，可以参考如下的代码：

```
public virtual ValidationResults ValidateForRegistration()
{
    var validator = ValidationFactory
           .CreateValidator<Customer>("IsValidForRegistration");
    var results = validator.Validate(this);
    return results;
}
```

上述方法会将列在 IsValidForRegistration 规则集中的验证程序应用到该类的指定实例(ValidateForRegistration 方法将成为你的实体类上的一个方法)。

注意：

有两种验证途径：有时，如果传递了无效数据你希望将其暴露出来；而有时，你想要收集错误并将其报告给代码的其他层。在.NET 中，代码约定是你会愿意纳入验证用途的另一项技术。然而，不同于批注和 VAB，代码约定只会检查条件，然后在第一次失败时抛出异常。你需要使用一个集中的错误处理程序以便从异常中恢复，并合理地降级运行。一般情况下，我会建议只在捕获潜在的可能导致不一致状态的严重错误时才在域实体中使用代码约定。例如，在工厂方法中使用代码约定就不错。这种情况下，如果向该方法传递明显无效的数据，你会想要它抛出一个异常。是否也在属性的设定方法中使用代码约定由你自己决定。我宁愿走温和路线，通过特性进行验证。

4.4　本章小结

输入表单在任何 Web 应用程序中都很常见，当然在 ASP.NET MVC 应用程序中也不例外。曾经有一段时间，业界有一种意见认为 ASP.NET MVC 并不适合用来支持数据驱动的应用程序，因为它们需要大量的数据输入和验证。最终，ASP.NET MVC 将任务完成地很好。它使用了一组与 Web Forms 不同的工具，但是也很有效和简明扼要。

ASP.NET MVC 为模板化帮助器提供了自动生成的重复性表单以及服务器端和客户端表

单的输入验证。如果构建自己的围绕以显示和验证特性批注的视图模型对象的用户接口，可在很大程度上简化输入验证。这些特性——称为数据批注——由 ASP.NET MVC 基础设施(模型元数据和验证提供程序)识别，并被处理以用于生成模板化帮助器并为用户反馈消息。

在这一章，我们几乎介绍了 ASP.NET MVC 编程的所有基础内容。第 5 章主要涉及一些待处理的问题，包括区域设置、安全和与内部对象的集成，主要是会话和缓存。

第 II 部分

ASP.NET MVC 软件设计

第 5 章

ASP.NET MVC 应用程序的特性

统治者过多全无益处。一国应当只有一位统治者，一位君王。

——*Homer*

Web 应用程序不仅仅只是一系列的请求/响应而已。如果真的如此简单的话，那么拥有一套成熟的 ASP.NET MVC 控制器，就已做好了万全准备。然而事实往往更复杂一些。一个有效的 ASP.NET MVC 应用程序来自于对方方面面的深刻考量和实现，包括搜索引擎优化(Search Engine Optimization，SEO)、状态管理、错误处理，以及本地化等(这里仅仅列示了一些比较重要的方面而已)。

这一章是截然不同且在某种程度上自包含的主题集合，每个主题都涉及市面上众多 ASP.NET MVC 应用程序已经具有或正考虑实现的一个特性。

注意：

不是每个 Web 应用程序都需要管理会话或全局状态。同样，并不是每个应用程序都是为国际用户所编写或者重视在搜索引擎中靠前的排名。然而话虽如此，我却想不出一个不需要所有这些特性的应用程序来。

5.1 ASP.NET 内部对象

ASP.NET MVC 是在 ASP.NET 基础架构之上运行和发展起来的。在很大程度上，可以把 ASP.NET MVC 视为经典 ASP.NET 运行时环境的特别版本，其仅用于支持不同的应用程序和编程模型。ASP.NET MVC 应用程序对构成 ASP.NET 生态系统的任何内置组件都具有完全访问权，包括 Cache、Session、Response，以及身份验证和错误处理层。

在 ASP.NET MVC 中访问这些组件的方式没有什么不同。但要从 ASP.NET MVC 内部使用这些组件呢？对这个问题的描述正是这一章的目的所在。

注意：

在 ASP.NET 中——包括这本书中——“内部对象”或是“内部”的表达常常用于表示封装在 HttpContext 中的整个基础对象集。包括 HttpRequest、HttpResponse、HttpSessionState、Cache 和用于标识已登录用户的对象。

5.1.1　HTTP 响应和 SEO

让我们从 HttpResponse 对象开始分析 ASP.NET 内部对象。该对象用来描述在请求处理结束时回传给浏览器的 HTTP 响应。HttpResponse 对象的公共接口可以让你设置 cookies 和内容类型、附加标头，以及针对响应数据的缓存将指令传递给浏览器。另外，HttpResponse 对象还有助于重定向到其他 URL。

要对响应流的任何一方面进行自定义，你需要写一个自定义操作结果对象，并让控制器方法返回你的类型的一个实例，而不是 ViewResult 或其他从 ActionResult 派生而来的预定义类型的实例。第 8 章“自定义 ASP.NET MVC 控制器”中会详细讨论这方面的内容。

SEO 是我们要更多关注 ASP.NET HttpResponse 对象特征的一个极其正当的理由。永久重定向是首个需要考虑的实际编程特性。

1. 永久重定向

在 ASP.NET 中，当你调用 Response.Redirect 时，会向浏览器返回一个 HTTP 302 代码，表明所请求的内容现在可以从另一个指定位置获得。在此基础上，浏览器会向指定地址发送第二个请求并获取内容。但是，访问你页面的搜索引擎会切实得到 HTTP 302 代码。HTTP 302 状态码的实际意思是所请求的页面已暂时移到新地址。其结果是，搜索引擎不会更新内部表，所以当以后某人单击查看你的页面时，引擎会返回原始地址。这样一来，浏览器就会收到一个 HTTP 302 代码，并需要发出第二个请求，以最终显示所需的页面。

注意：

要了解 HTTP 返回代码以及它们是如何工作的更多内容，请查看一个不错的资源 http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html。

如果重定向用于传送一个给定 URL 的请求，永久重定向就是一个更好的选择，因为它对于搜索引擎来说代表着一条更丰厚的信息。要设置永久重定向，需要返回 HTTP 301 响应代码。该代码会告知搜索代理，位置已被永久移动。搜索引擎知道如何处理 HTTP 301 代码，并使用这些信息来更新该页面 URL 的引用。当下一次它们显示涉及页面的搜索结果时，链接的 URL 就是新的。用这种方式，用户可以快速到达页面，避免了第二次往返。下面的代码显示了如何以编程方式设置一个永久重定向：

```
void PermanentRedirect(String url, Boolean endRequest)
```

```
{
    Response.Clear();
    Response.StatusCode = 301;
    Response.AddHeader("Location", url);
    ...

    // Optionally end the request
    if (endRequest)
       Response.End();
}
```

从 ASP.NET 4 开始，HttpResponse 类的特征便是对类似的事情采用一种新方法。该方法的名称是 RedirectPermanent。使用这种方法的方式与你使用经典的 Response.Redirect 方法相同，区别是现在调用程序会收到一个 HTTP 301 状态码。这对于浏览器的影响不大，但对于搜索引擎来说却是一个主要区别。

在 ASP.NET MVC 中，情况要容易得多，因为 RedirectResult 类型中添加了新的名为 Permanent 的 Boolean 成员。所有这些内容都封装在你很可能会使用的 Controller 类的 RedirectPermanent 方法中。

```
public ActionResult Index()
{
   ...
   return RedirectPermanent(url);
}
```

如前所述，你会在第 8 章了解到更多有关操作结果的高级自定义的话题。

2. 制定路由和 URL

ASP.NET Web Forms 与 ASP.NET MVC 之间的巨大差异在 ASP.NET MVC 中，URL 看起来更像是你发送给 Web 应用程序的命令，而非到页面和资源的服务器路径。因为路由机制是在开发人员的全面控制之下，所以你要负责为你的应用程序正确制定 URL。

如果认真思考 SEO，就会明白正确制定 URL 主要是指确保 URL 的唯一性。搜索引擎的主要目的之一是确定给定 URL 实际指向的内容的相关程度。当然，如果只能在一个地方并且通过唯一的 URL 找到一段给定的信息，那么该信息就具有更多的相关性。然而，有时候即便内容是唯一的，它也可以通过多个些许不同的 URL 找到。在这种情况下，你的风险是从搜索引擎获得的排名较低，更糟的是，对你网站该部分的引用会落在最后的结果页面，很难被潜在的访问者注意到。这里的问题与存储器和页面内容没多少关系，但与 URL 的格式有关系。即使万维网联盟(World Wide Web Consortium，W3C)建议你使用区分大小写的 URL，但是从 SEO 角度来看，使用单一大小写表示(且为小写)的 URL 是更好的选择。如果能保持所

有的 URL 都为小写，那么不仅能减少重复的 URL，还会加强网站的一致性。

注意：

与 Unix 系统不同，请求 URL 时 ASP.NET 和互联网信息服务(Internet Information Services，IIS)不区分大小写。大概是出于从 W3C 的角度考虑以及 Unix 的实践，搜索引擎将区分大小写的策略应用到了 URL 中。最近，大多数搜索引擎(主要是谷歌)都添加了各种形式的缓解措施，最终试图将这些区分大小写的 URL 也加入 URL 排名。无论如何，如果网站公开拼写不同的终结点，最终的排名绝对不只是各个不同排名的加总。所以，需要指出的底线就是，不注意使用单一大小写表示的 URL 在 SEO 排名中会相对靠后。

那么外部链接呢？

对于避免外部网站使用它们中意的大写或小写链接到你网站的页面，你可能无能为力。最有可能的情况是，他们直接复制你的 URL，由此重复你选择的大小写。如果不是你所选的大小写，你随时可以通过一个拦截 BeginRequest 事件的 HTTP 模块进行永久重定向。强制所有外部链接使用相同的大小写，会使你避免在多个 URL 间分化流量，以便将所有的流量专注于单个 URL 从而获取更高的排名(相对于其他软件方案中流行的“分而治之”的策略，我们称这种策略为“合而治之”)。

为了解决这一问题，还定义了标准的 URL 格式。标准的 URL 会以你心目中首选 URL 结构的形式来描述你所认为的 URL。只需要在<head>部分添加一个<link>标记，如下所示：

```
<link rel="canonical" href="http://myserver.com/" />
```

如果网站有大量的可以通过多个 URL 访问的内容，标准的 URL 就能为搜索引擎提供更多的信息，这样一来搜索引擎就可以将相似的 URL 作为同一个 URL 对待，由此产生对资源内容的更恰当的排名。标准 URL 特征(对于你来说是零成本的)的可能影响是它可以肃清有没有尾部斜杠的争论。标准的 URL 无论默认为哪种选择，对于搜索引擎来说实际的链接都没有区别。

3. 尾部斜杠

有一些与尾部斜杠有关的 SEO 问题。特别是在搜索引擎集成了一个筛选器，检测并惩罚性处理搜索结果中重复内容的情况下。重复的内容是指在搜索结果中被认为是与其他页面一样服务相同内容的页面(也就是说，任何不同的 URL)。从搜索引擎的角度看，带有尾部斜杠的 URL 和不带尾部斜杠的 URL 是提供相同内容的两个 URL。

要尽可能为用户提供最相关的内容，搜索引擎会尝试将看起来差不多一样的页面排名降低。但是，这一过程可能会无意中降低好网页的排名。

那么 ASP.NET MVC 和路由系统呢？应该强制有尾部斜杠吗？

最终，ASP.NET MVC 应用程序会完全负责其 URL 以及稍后的搜索引擎之请求。在一个新的应用程序中，这最终是由你决定的，因为你的路由决定了如何处理请求。用于在标记中生成 URL 的帮助器往往会避免尾部斜杠，所以我们说没有尾部斜杠是 ASP.NET MVC 中更为常见的解决方案。但是，请记住带有尾部斜杠的方法一样有效。在 ASP.NET MVC 中，用同样的方式解析(或不解析)带有和不带有尾部斜杠的 URL 也取决于你。你最终决定了你的网页排名。是否使用尾部斜杠并不像要符合你的选择那样重要。

如果要把现有网站移植到 ASP.NET MVC，可能会有很多遗留的 URL 要维护。可以安装一个自定义路由处理程序，将旧的 URL 永久重定向(HTTP 301)到新的 URL。这种方法很有效，但在实践中搜索引擎需要几周的时间物理更新链接的内部表，以反映所有的永久重定向。在此期间，你或许会失去一些收益。

搜索引擎总是喜欢处理现有的 URL。在这种情况下，你需要在微软 IIS 中安装一个重写模块，以将 ASP.NET MVC URL 映射到旧的 URL。

注意：

说到 SEO，还有一点需要说明，这就是子域，虽然这与页面或控制器操作并不严格相关。一般情况下，在网站的设置中有很多理由要考虑子域。常见的原因是将域拆分为子域，以便通过语言、产品或功能对网站进行区分，使得管理更为容易。但是，搜索引擎会把子域(包括裸域，比如 myserver.com)看作单独的网站。

5.1.2　管理会话状态

任何形态和形式的实际应用程序都需要维护自己的状态以服务用户请求。Web 应用程序也不例外。然而，不同于其他类型的应用程序，Web 应用程序需要特殊的系统级工具才能完成相同任务。这种特殊性的原因在于，Web 应用程序仍然依赖底层协议的无状态特性。只要 HTTP 仍然是 Web 的传输协议，所有的应用程序便会遇到同样的麻烦：找出最有效的方法来保存状态信息。

在 ASP.NET 中，HttpSessionState 类提供了基于字典的存储和检索会话状态值的模型。这个类不会在特定时间向操作应用程序的所有用户公开它的内容。只有发端于同一会话上下文的请求——即通过由同一浏览器实例所产生的多个页面请求而生成的——可以访问会话状态。会话状态能以多种方式存储和发布，包括在 Web 服务器场或 Web 园场景中。不过，默认会在 ASP.NET 工作进程中保持会话状态。

1. 使用 Session 对象

作为 ASP.NET MVC 开发人员，你在通向内部 Session 对象的路上并没有技术上的局限。你与 ASP.NET Web Forms 应用程序的开发人员有着差不多相同的利弊。ASP.NET MVC 基础

架构在内部使用会话状态，也可以在你的代码中这样做。特别是，ASP.NET MVC 基础架构会使用会话状态来保持 TempData 字典的内容，正如第 4 章“输入表单”中所述。

所以，如果觉得有必要跨会话存储数据，则先将其存储，然后通过熟悉的 Session 对象将其读取回来，如下所示：

```
public ActionResult Config()
{
  Session[StateEntries.PreferredTextColor] = "Green";
  ...
}
```

可以看出，一切与经典的 ASP.NET 编程没什么两样。但是要记住，会话字典是一个名称/值集合，因此它需要普通的字符串来标识条目。在代码中使用常量是防止发生致命错误的好方法。当你读取会话状态时，转换和 null 检查是不可避免的，如下所示：

```
var preferredTextColor = "";
var data = Session[StateEntries.PreferredTextColor];
if (data ! null)
preferredTextColor = (String) data;
```

在会话状态中存储的任何数据都会作为一个 Object 返回。

2. 绝不离开控制器

涉及访问 ASP.NET MVC 中会话状态的最重要一点是，只从控制器内部使用它。一般来说，在会话状态中存储的数据可以有两种使用方式。可以用来驱动一些输入数据的后端计算，也可以原样传递给视图。

在第二种情况下，确保你将单个值复制到视图模型类上的适当成员，或复制到任意一个你使用的预定义视图数据结构(ViewData 或 ViewBag)。从理论上说，也可以从 Razor 或 ASPX 视图中访问会话状态(及其他内部对象)。虽然所有像这样的代码都能运行，但你应该避免这样做，以保持控制器和视图之间较强的关注点分离(Separation of Concerns，SoC)。黄金法则是，视图从外部接收它需要集成的任何数据。

注意：

上述金科玉律对于呈现操作也同样适用，这些呈现操作就是你直接从视图调用的专门控制器方法。呈现操作会在你的视图中添加一点逻辑，但不破坏控制器和视图之间的分离——视图在控制器中将继续保持其唯一的联系。

5.1.3　缓存数据

缓存表明了应用程序将频繁使用的数据存储到中间存储介质的能力。在标准的 Web 场景

中，典型的中间存储介质是 Web 服务器的内存。但是，可以围绕每个应用程序的需求和特点来设计缓存，因此可以根据需要，使用任意层数的缓存达到你的性能目标。

在 ASP.NET 中，内置的缓存功能是通过 Cache 对象实现的。Cache 对象是在每一个 AppDomain 基础上创建的，并且在 AppDomain 运行期间它仍然保持有效。该对象自动清除内存和去除无用项的能力是其独有的。可以优先处理并将缓存项关联到各种类型的依赖项，比如磁盘文件、其他缓存项以及数据库表。当其中的任一项发生更改时，缓存的项会自动失效并删除。此外，Cache 对象提供了与 Session 一样常见的基于字典的编程接口。但与 Session 不同的是，Cache 对象不会在每个用户的基础上存储数据。

1. 本地 Cache 对象的优劣

在 ASP.NET MVC 中使用 Cache 对象与在 ASP.NET Web Forms 中使用是一样的。从控制器类或基础架构类，如 global.asax 中，访问 Cache 对象是比较合适的，如下所示：

```
protected void Application_Start()
{
  var data = LoadFromSomeRepository();
  Cache[CacheEntries.CustomerRecords] = data;
...
}
```

要读取缓存，可以使用刚刚看到的用于会话状态的相同模式，检查 nullness，并转换成一个已知的有效类型，如下所示：

```
var customerRecords = new CustomerRecords();
var data = Cache[CacheEntries.CustomerRecords];
if (data ! null)
customerRecords = (CustomerRecords) data;
```

如果缓存的数据需要在视图中显示，只需要将数据添加到特定视图的视图模型类即可。或者，可以在控制器类上定义呈现操作。

目前为止，一切都是正常的。那么在 ASP.NET MVC 中缓存的劣势是什么呢？

如前所述，Cache 对象受限于当前的 AppDomain 和随后的当前进程。这种设计在十年前还是不错的，但如今显示出了越来越多的局限性。如果要寻找一个类似于 Session 的全局存储库对象，跨 Web 服务器场或 Web 园架构工作，那么本地 Cache 对象是不适合的。你必须借助 Windows Server AppFabric Caching 服务，或是一些商业框架(如 ScaleOut 或 NCache)或者开源框架(比如 Memcached)。

然而，问题是 Cache 对象的实现并不基于与会话状态一样通用的提供程序模型。这意味着你需要将其扩展时，不能替换缓存数据的持有者，即使是测试控制器时也不行。

2. 注入缓存服务

目前在 ASP.NET MVC 中推荐使用的缓存方法包括在应用程序中注入缓存功能。定义一个抽象缓存服务的协议，并让你的控制器根据此协议工作。注入能够以你喜欢的方式进行，可以通过你喜欢的控制反转(IoC)框架，也可以通过一个特设的控制器构造函数的方式。后者被戏称为“穷人的依赖性注入方法”。

下面的代码显示了如何抽象缓存层的一个小而实用的示例：

```
public interface ICacheService
{
  Object Get(String key);
  void Set(String key, Object data);
  Object this[String key] { get; set; }
  ...
}
```

可以按照需求将该接口制作得丰富而巧妙。例如，你可能要添加成员以支持依赖项、过期事项和优先事项。请记住你不是在编写用于整个 ASP.NET 子系统的缓存层；而是在编写你的应用程序中的某一段代码。在这方面，YAGNI 原则(你不需要它)比以往任何时候都有用。

任何需要缓存的控制器类都将通过构造函数接受一个 ICacheService 对象，如下所示：

```
private ICacheService _cacheService;
public HomeController(ICacheService cacheService)
{
  _cacheService = cacheService;
}
```

接下来需要定义 ICacheService 接口的几个具体实现。具体类型会直接使用一种特定的缓存技术来存储和检索数据。下面是通过使用本地 ASP.NET Cache 对象实现该接口的一个类的框架：

```
public class AspNetCacheService : ICacheService
{
   private readonly Cache _aspnetCache;
   public AspNetCacheService()
   {
      if (HttpContext.Current != null)
         _aspnetCache = HttpContext.Current.Cache;
   }

   public Object Get(String key)
   {
```

```
        return _aspnetCache[key];
    }

    public void Set(String key, Object data)
    {
        _aspnetCache[key] = data;
    }

    public object this[String name]
    {
        get { return _aspnetCache[name]; }
        set { _aspnetCache[name] = value; }
    }
    ...
}
```

最后，让我们完成使用此缓存服务的控制器代码，以便能正确地注入该服务：

```
public class HomeController
{
    private readonly ICacheService _cacheService;
    public HomeController() : this(new AspNetCacheService())
    {
    }
    public HomeController(ICacheService cacheService)
    {
        _cacheService = cacheService;
    }
    ...
}
```

这样，需要缓存的控制器类就不会紧紧地绑定到某缓存对象的特定实现，而且最起码更易于测试。

3. 注入缓存服务的一个更好方式

往控制器实例中注入一个缓存服务要求必须为每个请求创建一个新的缓存服务。由于缓存服务是围绕现有及外部缓存数据持有者(比如 ASP.NETCache 对象或 Windows Server AppFabric Caching Services)的普通封装，因此不会对请求的性能有多大影响。事实上，缓存数据持有者只会在应用程序启动时初始化一次。

你能设法为应用程序的每个请求节省几个 CPU 周期并公开一个反过来基于可更换提供程序的全局缓存对象吗？当然可以。请尝试将下面的代码添加到 global.asax 中：

```
public class MvcApplication : HttpApplication
```

```
{
    ...

    // Internal reference to the cache wrapper object
    private static ICacheService _internalCacheObject;

    // Public method used to inject a new caching service into the application.
    // This method is required to ensure full testability.
    public void RegisterCacheService(ICacheService cacheService)
    {
       _internalCacheObject = cacheService;
    }

    // Use this property to access the underlying cache object from within
    // controller methods. Use this instead of native Cache object.
    public static ICacheService CacheService
    {
        get { return _internalCacheObject;  }
    }

    protected void Application_Start()
    {
        ...

        // Inject a global caching service
        RegisterCacheService(new AspNetCacheService());

        // Store some sample app-wide data
        CacheService["StartTime"] = DateTime.Now;
    }
}
```

这样，就不必在选定的控制器中为每个请求注入实际的缓存服务。缓存服务会在应用程序启动时一次性初始化并且注入。控制器使用应用程序对象上的公共静态方法(如在 global.asax 中定义的)来访问缓存。

```
var data = MvcApplication.CacheService[...]
```

RegisterCacheService 公共方法保留了可测试性。在任何你想要测试某个缓存识别控制器的单元测试中，都可以在单元测试的初步阶段放置下面的调用：

```
MvcApplication.RegisterCacheService(new FakeCacheService());
```

接着继续调用该控制器方法，它将显式使用该 FakeCacheService。

4. 分布式缓存

通过为缓存相关的任务使用公共协议，使其更易于测试，这并非你获得的唯一好处：通过让你的控制器随时识别缓存接口——并非缓存实现——它们便能够使用任何通过指定接口提供缓存服务的对象。换句话说，可以将上述的 AspNetCacheService 类替换为另一个依赖不同缓存架构的外形类似的类。

举例来说，可以依据分布式的架构，例如 Windows Server AppFabric Caching Services，或者另一个开源的或商业的框架，显式插入一个缓存服务。差不多所有这些框架都会公开一个公共 API，它类似于基本的 ASP.NET Cache 对象，因此不需要你做很多超出配置的设置工作。

这样很不错，但是如果对分布式缓存不感兴趣，也可以将 ASP.NET 本地 Cache 对象替换成最新的在.NET Framework 4 中引入的 MemoryCache 对象，它的明确目的就是为所有的.NET 应用程序提供缓存功能。为此，该类是在 ASP.NET 范围之外一个命名为 System.Runtime.Caching 的全新程序集中定义的。MemoryCache 对象类似于 ASP.NET Cache，不同之处在于如果尝试存储 null 值，它会引发异常。MemoryCache 类继承自基类 ObjectCache。通过派生你自己的缓存对象，可以控制内部存储以及缓存数据的管理。这并不是给大家的推荐方法，但它绝对是可以实现的。但是，请记住，ObjectCache 及其派生的类型并不适用于提供分布式缓存功能。如果想创建自己的分布式缓存，可能会面临在同步过程中维护多个缓存的麻烦。

5. 缓存方法响应

经典的 ASP.NET 输出缓存机制在 ASP.NET MVC 中同样适用。它采用 OutputCache 特性的形式，可以将其附加到控制器方法或控制器类，从而影响所有的操作方法。

```
 [OutputCache(Duration=10, VaryByParam="None")]
public ActionResult Index()
{
   ...
}
```

Duration 参数表示方法的响应应该在内存中保持缓存的时间(以秒为单位)。另一方面，VaryByParam 特性表示你应该缓存多少个不同版本的响应，每一个版本对应于指定属性的一个不同值。如果使用 None，就是告知系统你不希望同一个方法有多个版本的响应。

表 5-1 列出了该特性支持的属性。它们是 ASP.NET 的@OutputCache 指令特性的一个子集。未提及的特性是局限于 ASP.NET 用户控件的那些特性。

表 5-1　OutputCache 特性的属性

特　　性	描　　述
CacheProfile	将响应与 web.config 文件中指定的一组输出缓存设置关联起来
Duration	响应缓存的时长(以秒计)
Location	指定存储方法调用响应的位置(浏览器、代理或服务器)。该特性的值是从 OutputCacheLocation 枚举中获取的
NoStore	表示是否发送一个 Cache-Control:no-store 标头来阻止浏览器端的响应存储
SqlDependency	表示微软 SQL Server 数据库上指定表的一个依赖项。在表内容变更时，响应会从缓存中移除
VaryByContentEncoding	你要用于区分缓存响应的内容编码
VaryByCustom	一个用分号分隔的字符串列表，通过该列表可以基于浏览器类型或用户定义的字符串来维持不同的响应缓存副本
VaryByHeader	用分号分隔的 HTTP 标头列表
VaryByParam	一个用分号分隔的字符串列表，该列表表示的是用 GET 方法特性发送的查询字符串的值，或者用 POST 方法发送的参数

这些属性传递的是相同的 ASP.NET 运行时的输出缓存的基础架构，与在 ASP.NET Web Forms 中的运行情况完全一样。

注意：

这应该是显而易见的，但让我们说得更清楚一些吧。你设置 OutputCache 特性的方法在提供了有效的缓存响应时，不会为响应到达服务器的请求而执行。

6. 局部输出缓存

局部缓存并不局限于方法的整体响应。你还可以将 Output Cache 特性附加到子操作。子操作是控制器上的一个方法，视图可以通过使用 Html.RenderAction 帮助器回调它。RenderAction 帮助器可以调用控制器上的任何方法；但某些方法可以标记为独占的子操作。通过使用如下所示的 ChildActionOnly 特性来实现这一点：

```
[ChildActionOnly]
public ActionResult RenderSiteMap()
{
   ...
}
```

这种方法显然用于呈现视图的一小部分。在逻辑上等同于 ASP.NET Web Forms 中的用户

控件。通过使用 OutputCache 特性修饰该方法，就可以按照指定持续时间缓存响应了。

5.2　错误处理

由于 ASP.NET MVC 是在经典的 ASP.NET 运行时环境之上运行的，所以你不可能期望找到一个完全不同的处理运行时错误的基础架构。这就是说可以继续选择经典的 ASP.NET 策略，将任何 400 或 500 的 HTTP 状态代码映射到提供错误信息的特定 URL。在错误需要 HTTP 重定向的情况下以编程方式切换到另一个页面。可以通过 web.config 文件的<customErrors>节来控制映射。

依我之见，这种方法虽然可用，但不如 ASP.NET MVC 中的方法理想，可以通过直接改变由控制器调用的视图模板的名称，轻松地切换到错误界面。

让我们看看当 ASP.NET MVC 遇到错误处理的时候所要提供的内容。总体来看，ASP.NET MVC 中的错误处理跨越了两个主要领域：程序异常和路由异常的处理。前者是关于在控制器和视图中捕获错误的；而后者更多是有关重定向和 HTTP 错误的。

5.2.1　处理程序异常

你在 ASP.NET MVC 应用程序中编写的代码大部分留存于控制器类中。在控制器类中，可以用多种等效的方式处理可能的异常：可以直接通过 try/catch 块、通过重写 OnException 方法、也可以使用 HandleError 特性来处理异常。

1. 直接处理异常

首先，可以使用本地的 try/catch 块来防止代码特定部分的可能异常。这种方式为你提供了最大的灵活性，但代价是它会给代码带来些许干扰。我并不质疑这种异常处理的重要性——顺便说一句，这是错误处理的官方.NET 方法——但是 try/catch 块的存在会使代码的阅读变得困难。出于这个原因，我总是会欣然接受那些将异常处理的代码集中到合理范围内的替代解决方案。

要执行一段确定能够捕获引起任何(或只是一些)异常的代码，可以使用下面的代码：

```
try
{
   // Your regular code here
   ...
}
catch
{
   // Your recovery code for all exceptions
```

```
    ...
}
```

上面的代码段会捕获由 try 块中的代码所引起的各种异常。由于其极大的通用性，它不会将你置于实现有效恢复策略的位置。示例代码段可以有很多变化和扩展。例如，可以列出多个 catch 块，每个显著的例外对应一个。你还可以添加一个 finally 块，它将最终完成操作和运行，而不考虑执行流通过 try 块还是 catch 块：

```
try
{
    // Your regular code here
    ...
}
catch(NullReferenceException nullReferenceException)
{
    // Your recovery code for the null reference exception
    ...
}
catch(ArgumentException argumentException)
{
    // Your recovery code for the argument exception
    ...
}
finally
{
    // Finalize here, but DON'T throw exceptions from here
    ...
}
```

从最具体到最概略的异常情况都列示出来了。从 catch 块中，你还可以处理异常，以便其他顶层模块不会获知相关情况。或者，可以妥善地处理这种情况并回收。最后，可以做一些工作，然后重新抛出相同的异常或设置一个新的异常，其中带有一些额外的或修改过的信息。

涉及编写处理异常的直接代码时，你还需要牢记几条原则。首先，如果代码运行到 catch 块，其代价是极其高昂的。因此，你应该明智地使用它——只在确实需要且不会过度捕获的时候使用。

注意：

长时间以来，微软都建议从 System.ApplicationException 派生异常类。最近，这一观点出现了彻底的转变：新的指令显示出相反的情况。应该忽略 ApplicationException，并从 Exception 或其他更特定的内置类来派生你的异常类。同时，别忘了让你的异常类序列化。更多的背景

信息，请参阅 StackOverflow 的以下线程(以及所包含的链接)http://stackoverflow.com/questions/5685923/what-is-applicationexception-for-in-net。

此外，一定不要引发作为根 System.Exception 类实例的异常。强烈建议你尝试使用内置的异常类型，如 InvalidOperationException、NullReferenceException 和 ArgumentNullException 等，只要这些类型适用。你应该避免全面使用自己定义的异常，虽然对于程序错误，你应该考虑定义自己的异常。一般情况下，应当将异常具体化。ArgumentNullException 就比 ArgumentException 更具体。异常通常会带有一条消息，该消息必须针对开发人员，并且在理想情况下要本地化。

注意：

来自微软的关于异常处理的另一项主要原则是，对于一般的错误(比如 null 引用、无效的参数、I/O 或网络异常)使用内置类型，对于特定于你正在创建的应用程序，要创建应用程序专属的异常类型。

2. 重写 OnException 方法

正如第 1 章“ASP.NET MVC 控制器”中所讨论的，各个控制器方法的执行都受到一个被称作操作调用程序的特殊系统组件的调控。关于默认操作调用程序有意思的一点是，它会一直在 try/catch 块中执行控制器方法。下面是一些说明默认操作调用程序行为的伪代码：

```
try
{
    // Try to invoke the action method
    ...
}
catch(ThreadAbortException)
{
    throw;
}
catch(Exception exception)
{
   // Prepare the context for the current action
   var filterContext = PrepareActionExecutedContext( ..., exception);

   // Go through the list of registered action filters, and give them a chance
      to recover
   ...

   // Re-throw if not completely handled
   if (!filterContext.ExceptionHandled)
```

```
    {
        throw;
    }
}
```

如果在方法的执行过程中或在视图的呈现过程中的某一时刻引发了一个异常，该控件就会运行 catch 块中的代码，只要该异常不是 ThreadAbortException。处理异常需要循环遍历已注册的操作筛选器列表，并把解决问题的机会留给它们自己。在循环结束时，如果异常未被标记为已处理，则捕获到的异常会再一次被抛出。

操作筛选器是一段代码，它可以被注册用来处理一些在操作方法执行过程中所引发的事件。其中的一个系统事件就是在调用程序截获到异常的时候触发(我会在第 8 章对操作筛选器做详细讲解)。将你自己的代码添加到筛选器列表的最简单方法是重写控制器类上的 OnException 方法，如下所示：

```
protected override void OnException(ExceptionContext filterContext)
{
  ...
}
```

该方法可以在你的所有控制器类(或你的一个基类)中定义，且常常在操作方法发生未处理异常时被调用。

注意：

异常是不会由源于控制器范围之外的 OnException 捕获的，比如来自模型绑定层失败中的 null 引用，或从无效路由中所产生的未发现异常。我会在本章稍后详细探讨这一方面的内容。

总体来看，只有一个理由重写控制器类中的 OnException，即你希望控制系统的行为以及在异常情况下平稳降级运行。这意味着 OnException 中的代码被赋予了对于失败请求的整个响应的控制权。这个方法会接收一个 ExceptionContext 类型的参数。该类型带有一个 ActionResult 类型的 Result 属性。你可能会猜到，这个属性指的是下一个视图或操作结果。如果 OnException 中的代码省略了设置结果，那么用户就不会看到任何错误界面(不论是系统的还是应用程序的错误界面)；用户只会看到一个空白界面。下面是实现 OnException 的典型方式：

```
protected override void OnException(ExceptionContext filterContext)
{
    // Let other exceptions just go unhandled
    if (filterContext.Exception is InvalidOperationException)
    {
```

```
        // Default view is "error"
        filterContext.SwitchToErrorView();
    }
}
```

SwitchToErrorView 方法是用于 ExceptionContext 类的扩展方法，其编码如下所示：

```
public static void SwitchToErrorView(this ExceptionContext context,
                               String view = "error", String master = "")
{
    var controllerName = context.RouteData.Values["controller"] as String;
    var actionName = context.RouteData.Values["action"] as String;
    var model = new HandleErrorInfo(context.Exception, controllerName,
                actionName);
    var result = new ViewResult
                 {
                     ViewName = view,
                     MasterName = master,
                     ViewData = new
                            ViewDataDictionary<HandleErrorInfo>(model),
                     TempData = context.Controller.TempData
                 };
    context.Result = result;

    // Configure the response object
    context.ExceptionHandled = true;
    context.HttpContext.Response.Clear();
    context.HttpContext.Response.StatusCode = 500;
    context.HttpContext.Response.TrySkipIisCustomErrors = true;
}
```

总的来说，这些代码提供了一个有效的框架级 try/catch 块，它不仅捕获异常，还会切换到一个错误视图。在上面所示的代码中，默认的错误视图是 error，但可以对它及其布局进行更改。

3. 使用 HandleError 特性

作为重写 OnException 方法的替代选项，可以用 HandleError 特性以及由它派生出来的自定义类来修饰这个类(或个别的方法)。

```
 [HandleError]
public class HomeController
{
    ...
```

```
}
```

注意，HandleError 比一个简单特性的含义要多一些；它是一个操作筛选器。就其本身而言，它包含可执行代码，而且并不局限于向其他一些模块提供元信息。尤其是，HandleError 实现了 IExceptionFilter 接口，如下所示：

```
public interface IExceptionFilter
{
   void OnException(ExceptionContext filterContext);
}
```

该接口也是所有控制器实现的接口。但是，在控制器的基类上，OnException 只有一个空主体。

在内部，HandleError 通过使用一段非常类似于 SwitchToErrorView 的代码实现了 OnException 方法。唯一的区别是操作视图更改的情形。HandleError 特性只有在它们没有被提前完全处理且并不从子操作中产生的情况下，才会捕获指定的异常。若要控制你要处理的异常，请遵循下面的做法：

```
[HandleError(ExceptionType=typeof(NullReferenceException),
   View="SyntaxError")]
```

每个方法都可以有多个特性匹配项，每一个匹配项对应一个相关的异常。View 和 Master 属性表示要在异常后显示的视图。默认情况下，HandleError 会切换到名为 error 的视图(这样一个视图是由微软 Visual Studio ASP.NET MVC 标准模板特意创建的)。

重要提示：

为了在调试模式下产生 HandleError 的任何可见结果，你需要打开应用程序级别的自定义错误，如下所示：

```
<customErrors mode="On">
</customErrors>
```

如果对配置文件的<customErrors>节进行了默认设置，那么只有远程用户会得到所选的错误页面。本地用户(比如正在进行调试的开发人员)将收到经典的错误页面，其中带有由普通 ASP.NET 异常处理程序所生成的有关堆栈跟踪的详细信息。

小贴士：

即使所有一切都完全配置为显示自定义错误页面，Internet Explorer 显示内置错误页面的情况也可能发生。这归因于自 2006 年以来就已经存在的一个不为人知的 Internet Explorer 功能。在实际情况中，如果 Internet Explorer 检测到你的错误页面有不大于 512 个字节的正文，它仅仅会直接显示一个内置页面，因为——或者它相信——这让用户看到会更好！这对于现

实中的网站和网页算不了什么，但在开发的初始阶段，它却很容易令人头疼。

就像 ASP.NET MVC 中的其他任何操作筛选器一样，可以通过将 HandleError 特性注册为全局过滤器，从而将其自动应用到任意控制器类的方法上。顺便说一句，这正是用于 ASP.NET MVC 的 Visual Studio 模具所生成的代码内部发生的情况。下面是来自 global.asax 中的 Application_Start 的一段摘录：

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        RegisterGlobalFilters(GlobalFilters.Filters);
        ...
    }

    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }
}
```

全局筛选器是默认的操作调用程序在调用操作方法之前自动添加到筛选器列表中的一种操作筛选器。我还会在第 8 章中对全局操作筛选器做更多介绍。

5.2.2　全局错误处理

大多数情况下，OnException 和 HandleError 特性会提供面向错误处理的控制器级别的控制。这意味着，每个需要错误处理的控制器必须承载一些异常处理代码。但更重要的是，在控制器级别处理错误并不保证能够拦截围绕你应用程序所引发的所有可能异常。

可以创建一个应用程序级别的错误处理程序，来捕获所有未处理的异常，并将它们路由到指定的错误视图。

1. 来自 global.asax 的全局错误处理

从 ASP.NET 运行时的第一个版本开始，HttpApplication 对象——global.asax 背后的对象——就以 Error 事件为特点了。每当未处理的异常到达应用程序中代码的最外层时即引发该事件。下面是如何编写这样一个处理程序的例子：

```
void Application_Error(Object sender, EventArgs e)
{
  ...
}
```

可以在此事件处理程序中做一些有用的事情，例如向网站管理员发送电子邮件，或写入微软 Windows 事件日志以记录页面未能正确执行的信息。下面是一个示例：

```
void Application_Error(Object sender, EventArgs e)
{
    var exception = Server.GetLastError();
    if (exception == null)
        return;

    var mail = new MailMessage { From = new MailAddress
                                     ("automated@contoso.com") };
    mail.To.Add(new MailAddress("administrator@contoso.com"));
    mail.Subject = "Site Error at " + DateTime.Now;
    mail.Body = "Error Description: " + exception.Message;
    var server = new SmtpClient { Host = "your.smtp.server" };
    server.Send(mail);

    // Clear the error
    Server.ClearError();

    // Redirect to a landing page
    Response.Redirect("home/landing");
}
```

有了前面的代码，管理员就会收到一封电子邮件(如图 5-1 所示)，但用户仍然会收到一个系统错误页面。如果要避免这种情况，可以在你处理错误之后将用户重定向到一个着陆页面。最后要明确的是，如果 SMTP 服务器需要身份验证，那就需要通过 SmtpClient 类的 Credentials 属性提供你的凭据。

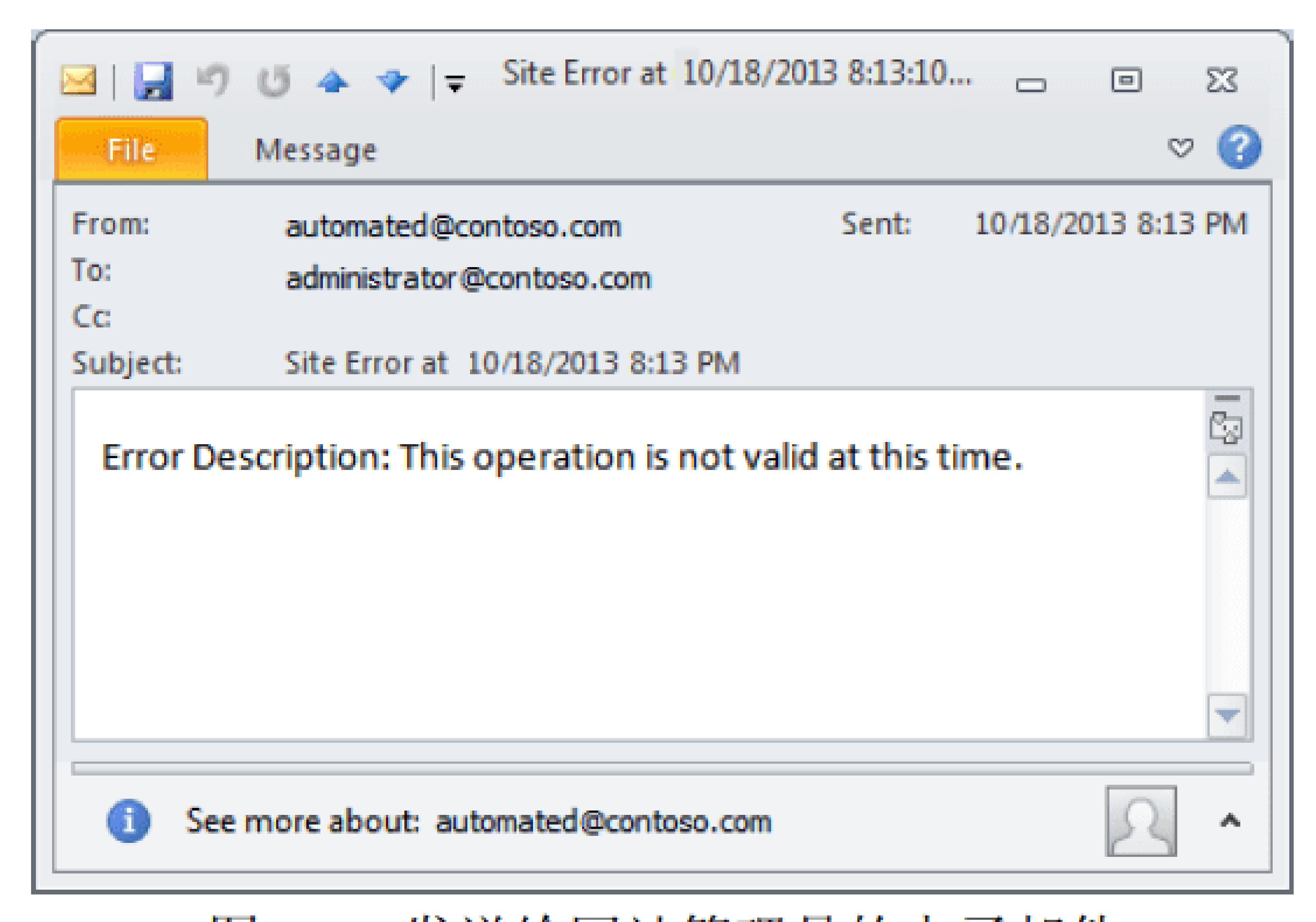


图 5-1　发送给网站管理员的电子邮件

2. 使用 HTTP 模块的全局错误处理

在 ASP.NET 中，捕获致命的异常只有一个方法：即编写一个用于 HttpApplication 对象 Error 事件的处理程序。但有两种实现方式。可以直接在应用程序的 global.asax 文件中编写代码，也可以在 web.config 文件中插入一个量身定做的 HTTP 模块。HTTP 模块会注册自己的用于 Error 应用程序事件的处理程序。这两种解决方案在功能上相同，但基于 HTTP 模块的解决方案可以在无须重新编译应用程序的情况下打开、关闭以及修改。它是一种不那么烦人的方式。

在考虑一个全局错误处理程序时，你需要牢记两个理念：向管理员警示异常以及日志记录异常。特别是对于第二个任务，HTTP 模块似乎是比 global.asax 中的代码更容易管理的解决方案。

在 ASP.NET 开发人员之间比较流行的工具是错误日志记录模块和处理程序(Error Logging Modules And Handlers，ELMAH)。ELMAH 本质上是由一个 HTTP 模块构成，一旦配置，就会拦截应用程序级别的 Error 事件，并根据大量后端存储库的配置将其记录下来。ELMAH 源自一个开源项目(http://code.google.com/p/elmah)以及大量的扩展，这些扩展主要是在存储库方面进行的。ELMAH 提供了一些很好的设施，比如可以在上面查看所有记录过的异常并对每一个异常进行仔细研究的网页。从体系结构上来说，任何专属于 ASP.NET 的错误报告系统都与 ELMAH 区别不大。

3. 拦截模型绑定的异常

集中式的错误处理程序还善于捕获源自控制器以外的异常，比如由不正确的参数引起的异常。如果声明了带有一个参数的控制器方法，例如整数参数，而当前绑定器不能匹配任何提交到该参数的值，就会得到一个异常。从技术上讲，该异常并不是由模型绑定器本身所触发，而是由一个操作调用程序的内部组件在准备方法调用时引发的。如果该组件找不到用于非可选方法参数的值，它就会抛出一个异常。

此异常不会从控制器代码的内部抛出，但它依然在操作调用程序中处于整体 try/catch 块的控制之下。全局(或本地)的 HandleError 为什么不进行异常捕获呢？它会的，但只有在你打开了 web.config 文件中的自定义错误标志时才会。当自定义错误标志关闭时，拦截模型绑定错误的唯一办法就只能是借助于 global.asax 中的集中式错误处理程序。图 5-2 显示了妥善处理后提供给用户的页面，以及发送给管理员的电子邮件消息。

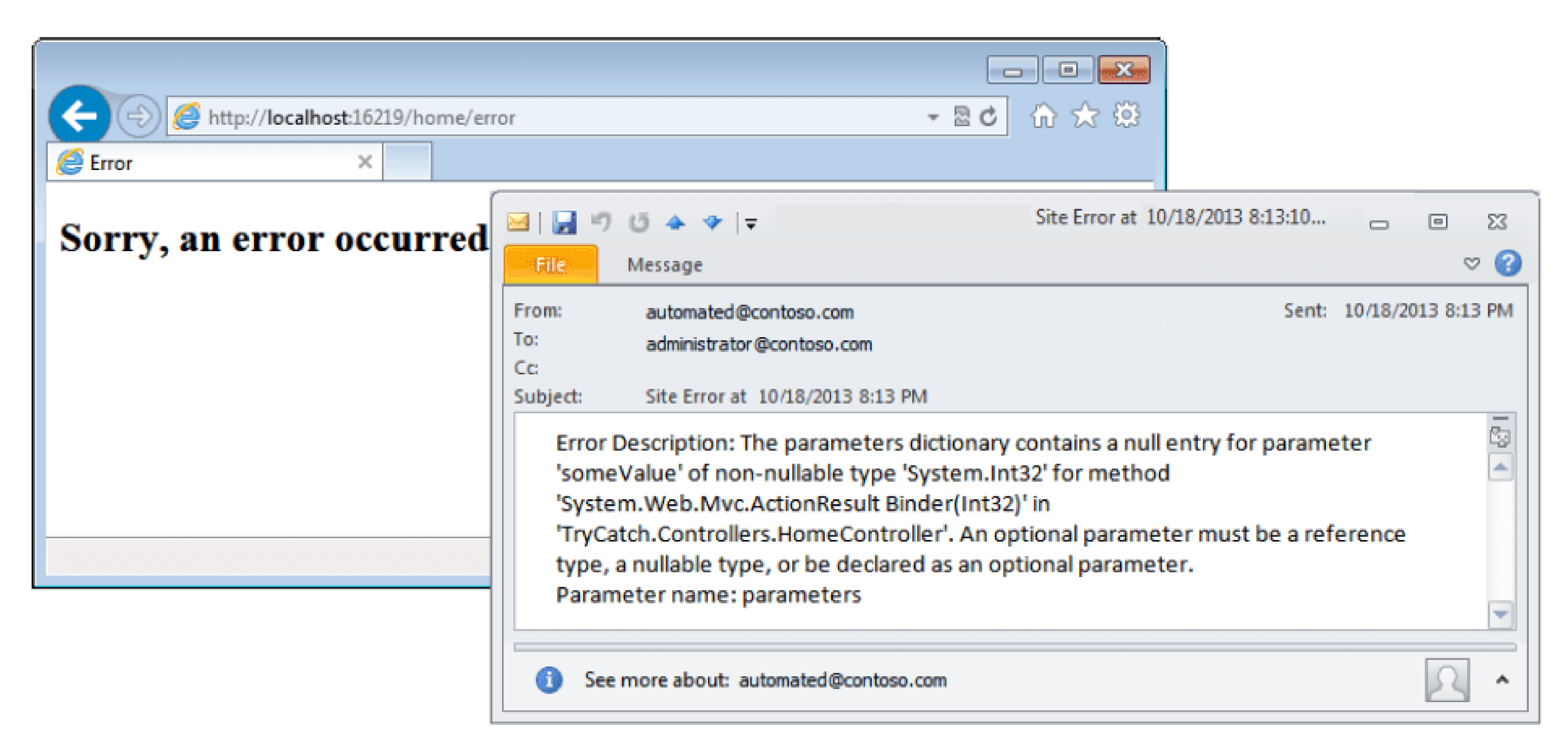


图 5-2　由 Application_Error 处理程序捕获的模型绑定器异常

4. 处理路由异常

除了检测到的程序错误，你的应用程序还可能因为传入请求的 URL 不匹配任何映射的路由而抛出异常——无论是由于无效的 URL 模式(无效的操作或控制器名称)还是违反了约束。在这种情况下，你的用户会收到一个 HTTP 404 错误。你可能有多种原因希望能避免让用户接收默认的 404 ASP.NET 页面，但最主要的是为了向最终用户提供更友好的页面。

由 ASP.NET 框架强制执行的典型解决方案包括为 404 和 403 等常见的 HTTP 代码定义的自定义页面(或 ASP.NET MVC 中的路由)。每当用户输入或打开一个无效的 URL 时，他就会被重定向到另一个提供了一些有用信息(希望如此)的页面。以下是如何在 ASP.NET MVC 中注册特定路由：

```
<customErrors mode="On">
<error statusCode="404" redirect="/error/show" />
...
</customErrors>
```

这一招很管用，从纯粹功能的角度来说没有什么可质疑的。那么，问题到底在哪里呢？

第一个问题是安全性。通过将 HTTP 错误映射到个性化的视图，黑客可以区分应用程序中可能发生的不同类型的错误，并利用这些信息计划进一步的攻击。因此，你要将<customErrors>节的 defaultRedirect 特性设置成指定并且固定的 URL，并确保未设置各状态代码。

各状态代码视图的第二个问题与 SEO 有关。想象一下，一个搜索引擎在一个实现自定义错误路由的应用程序中请求一个并不存在的 URL。应用程序首先会生成一个 HTTP 302 代码，并通知调用方资源已暂时迁至另一个位置。这时，调用方会做第二次尝试，并终于到达错误页面。这种方法对人来说不错，能最终得到一条漂亮的消息；但从 SEO 的角度来看算不上最

优，因为它会导致搜索引擎断定内容没有遗漏，只是比平常更难检索而已。并且，错误页面会作为常规内容编目到相关类似的内容。

另一方面，路由异常是一种特殊类型的错误，并且值得区别于程序错误的特别策略。基本上，路由异常都涉及一些缺失的内容。

5.2.3　处理缺失内容

路由子系统是应用程序的前端，也是请求 URL 获取一些内容的入口。在 ASP.NET MVC 中，以与处理有效请求相同的方式处理缺失内容的请求很简单。如果创建一个捕获将不被处理的所有请求的专用控制器，就不需要重定向和额外的配置。

1. 全部捕获(catch-all)路由

处理这种情况的常见做法是在 global.asax 中完成路由集合，用一个全部捕获(catch-all)路由捕获发送到你的应用程序的尚未被任何现有路由所捕获的 URL。

```
public static void RegisterRoutes(RouteCollection routes)
{
    // Main routes
    ...

    // Catch-all route
    routes.MapRoute(
      "Catchall",
      "{*anything}",
      new { controller = "Error", action = "Missing" }
    );
}
```

很明显，全部捕获规则需要在路由列表的最底部运行。这是很必要的，因为路由的计算是从顶部到底部，且分析会止于找到第一个匹配项的时候。全部捕获路由会将请求映射到你的应用程序专属的 Error 控制器(在 ASP.NET MVC 中并没有现成的 Error 控制器，但强烈建议你自己创建一个)。反过来，Error 控制器着眼于内容和标头，并随后决定要返回的 HTTP 代码。下面是这种 Error 控制器的一个示例：

```
public class ErrorController : Controller
{
    public ActionResult Missing()
    {
      HttpContext.Response.StatusCode = 404;
      HttpContext.Response.TrySkipIisCustomErrors = true;
```

```
        // Log the error (if necessary)
        ...

        // Pass some optional information to the view
        var model = ErrorViewModel();
        model.Message = ...;
        ...

        // Render the view
        return View(model);
    }
}
```

上述示例中的 ErrorViewModel 类是你以强类型方式用于将数据传递到基础视图的任意视图模型类。使用 ViewData 字典也可以，且总的来说 ViewData 字典是这个特定且相对简单的上下文中一个可接受的折中方案。

通过使用错误控制器，可以提高应用程序的友好性，并为搜索引擎对其进行优化。实际上，在将直接的(即非重定向)错误代码返回给任意调用方的同时，你也为用户提供了一个漂亮的用户界面。

注意：

全部捕获路由就是一个为 URL 选出的不匹配任何其他路由的路由。但是，许多路由是通过标准路由来匹配的，该标准路由总体来说是一个相当通用的几乎全部捕获(catch-almost-all)的路由。换言之，像/foo 这样的一个 URL 会匹配默认路由，并且永远不会到达全部捕获路由。因此，如果它缺少一个 Foo 控制器，就会导致 404 错误。为了拦截因无效控制器名称而导致的 404 错误，你需要重写控制器工厂(第 7 章还将就此问题讨论更多细节)。

2. 跳过 IIS 错误处理策略

在前面的代码片段中，ErrorController 类上的 Missing 方法在某些时候会将 Response 对象的 TrySkipIisCustomErrors 属性设置为 true。它是在 ASP.NET 3.5 中引入的属性，专门处理 IIS 7 集成管道的一个功能。

当一个 ASP.NET 应用程序(Web Forms 或 ASP.NET MVC)在 IIS 7 下的集成管道内运行时，一些 ASP.NET 的配置设置会与在 IIS 级别定义的设置合并，如图 5-3 所示。

尤其是，如果在 IIS 中定义用于常见 HTTP 状态代码的错误页面，默认情况下这些页面将优先于由 ASP.NET 生成的内容。其结果是，你的应用程序可能会捕获一个 HTTP 404 错误，并向用户提供一个好看的特设网页。不论愿意与否，你的网页都不会到达最终用户，因为它会被在 IIS 级别设置的另一个页面所取代。

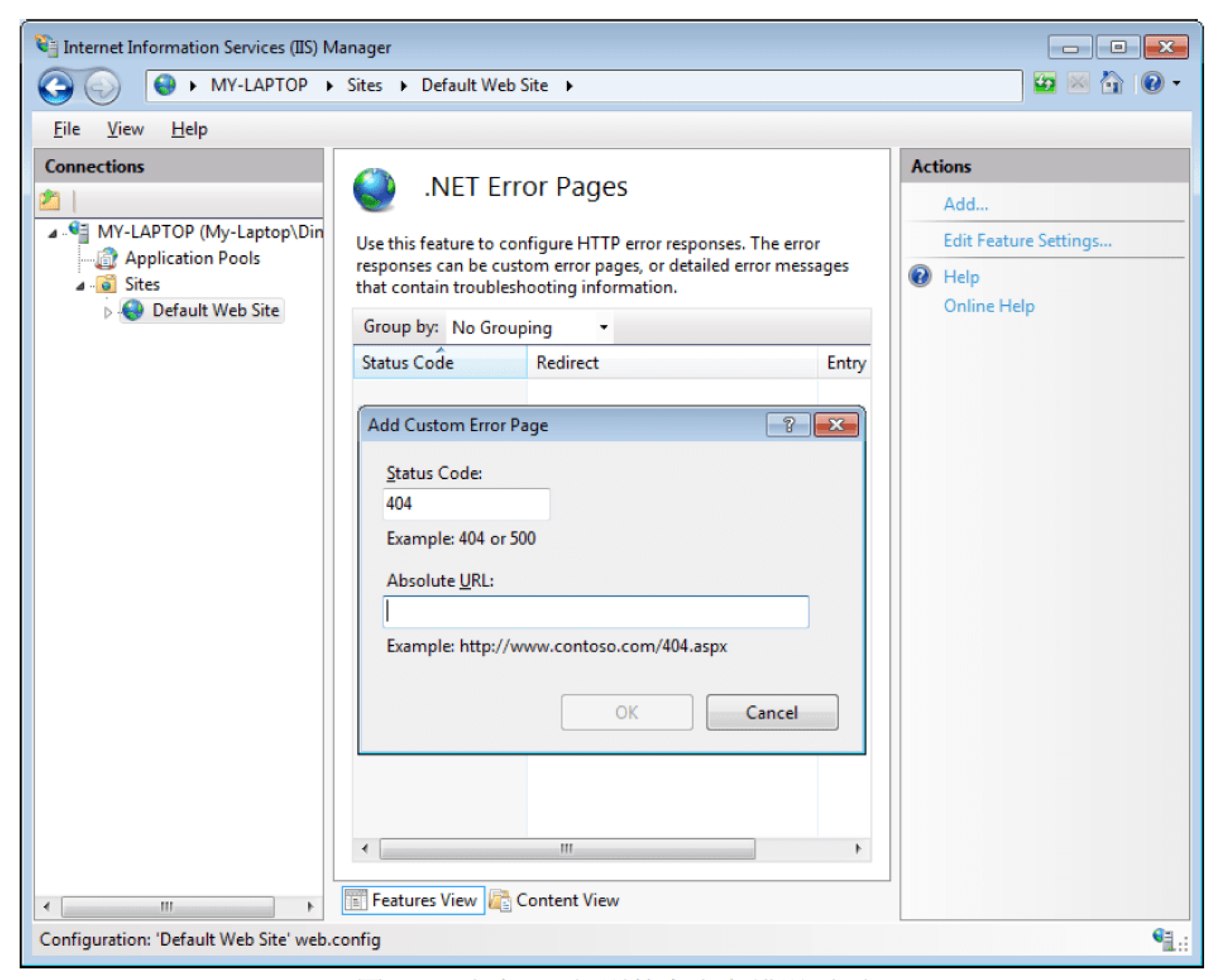


图 5-3　定义 IIS 级别的自定义错误页面

为了确保绕过 IIS 的错误处理，请将 TrySkipIisCustomErrors 属性设置为 true。该属性只对在 IIS 7 集成管道模式下运行的应用程序有用。在集成管道模式下，该属性的默认值是 false。例如，HandleError 异常筛选器的实现，就会仔细考虑这一特性，并将该属性设置为 true。

5.3　本地化

本地化的整个主题在.NET Framework 中并不算新鲜，在 ASP.NET 中也不例外。从 ASP.NET 的第一个版本开始，就有工具编写区域特性的页面了。其好处就在于一切都没有改变，因此将本地化功能添加到 ASP.NET MVC 应用程序不会比在经典的 ASP.NET 中更难，也不会有太大的差别。

从预期寿命不短的整个应用程序的角度来考虑本地化，有三个方面的问题需要解决：如何让(所有)资源本地化，如何添加对新区域的支持，以及如何将数据库用作(或是否将数据库用作)本地化信息的存储位置。

5.3.1　使用可本地化的资源

资源就是你希望适应于特定区域性的用户界面项。资源跨越整个应用程序，并不限于要翻译的文本。我建议忘记老式的 ASP.NET Web Forms 的做法，勾勒出你自己的特定于现代 Web 应用程序的资源管理策略。这种策略基于两大支柱，即：RESX 资源文件的组织和内容的粒度。

RESX 文件的作用是什么？

RESX 文件归根结底是由 Visual Studio 设计器在运行中编译的 XML 文件。作为开发人员，你能够部分控制名称空间，还能访问类成员的修饰符。换句话说，当你在项目中添加一个资源时，可以选择是否让所有的属性公开化或者内部化(默认情况)，并决定哪个名称空间对其进行分组。如果在它们自己的程序集中编译资源，那么公共修饰符就是必要的。

1. 可本地化的文本

可本地化的文本是依然可以便利地留在经典 RESX 文件中的唯一一种资源。从我在实战中所总结的经验来看，用单个全局文件来存放所有可本地化的资源变成了一种不太愉快的体验，即便对一般复杂的应用程序来说也是如此。

问题之一是文件大小增长得太快；问题之二更令人苦恼，多个开发人员可能会在同一个文件上做并发编辑，随后还需要做连续的合并。但是，我支持你不要忽略命名问题。当有几百个涵盖整个应用程序范围的字符串时，你该如何对其命名？许多字符串看起来相同，或仅在细微之处存有不同。许多字符串出于某些合理的原由不是完整的字符串；它们往往是位数以及一些需要由动态生成的内容去完成的文本片段。相信我：在只有一些页面的限制内容的条件下对其中一些进行命名是可行的；但要处理用于整个应用程序的数以百计的命名，的确十分困难。

总的来说，最好的办法是使用多个资源文件。理想情况下，你会希望每个视图有一个 RESX 文件。这不一定会导致文件的泛滥：对于一个复杂度最小的视图，单独的 RESX 文件会使本地化的过程容易得多，且更易于管理。

至于文件分组，我建议创建一个以调用视图的控制器命名的子文件夹。具体来说，这意味着在你的项目中创建一个 Resources 文件夹，其中包含分布在一堆依据某些标准所组织的子文件夹中的多个 RESX 文件。可能为每个逻辑功能区域分配一个文件，或者更好的情况可以是每个控制器一个。一个 Shared 子文件夹可能有助于将资源文件以及从多个视图中引用的内容收集到一起。图 5-4 显示了一个对多个文件进行分组的 Resources 文件夹。

放置在这样的 RESX 文件中的字符串都是全局性的，并且可以从任何视图和控制器类中引用。下面是 Razor 中所需的代码段：

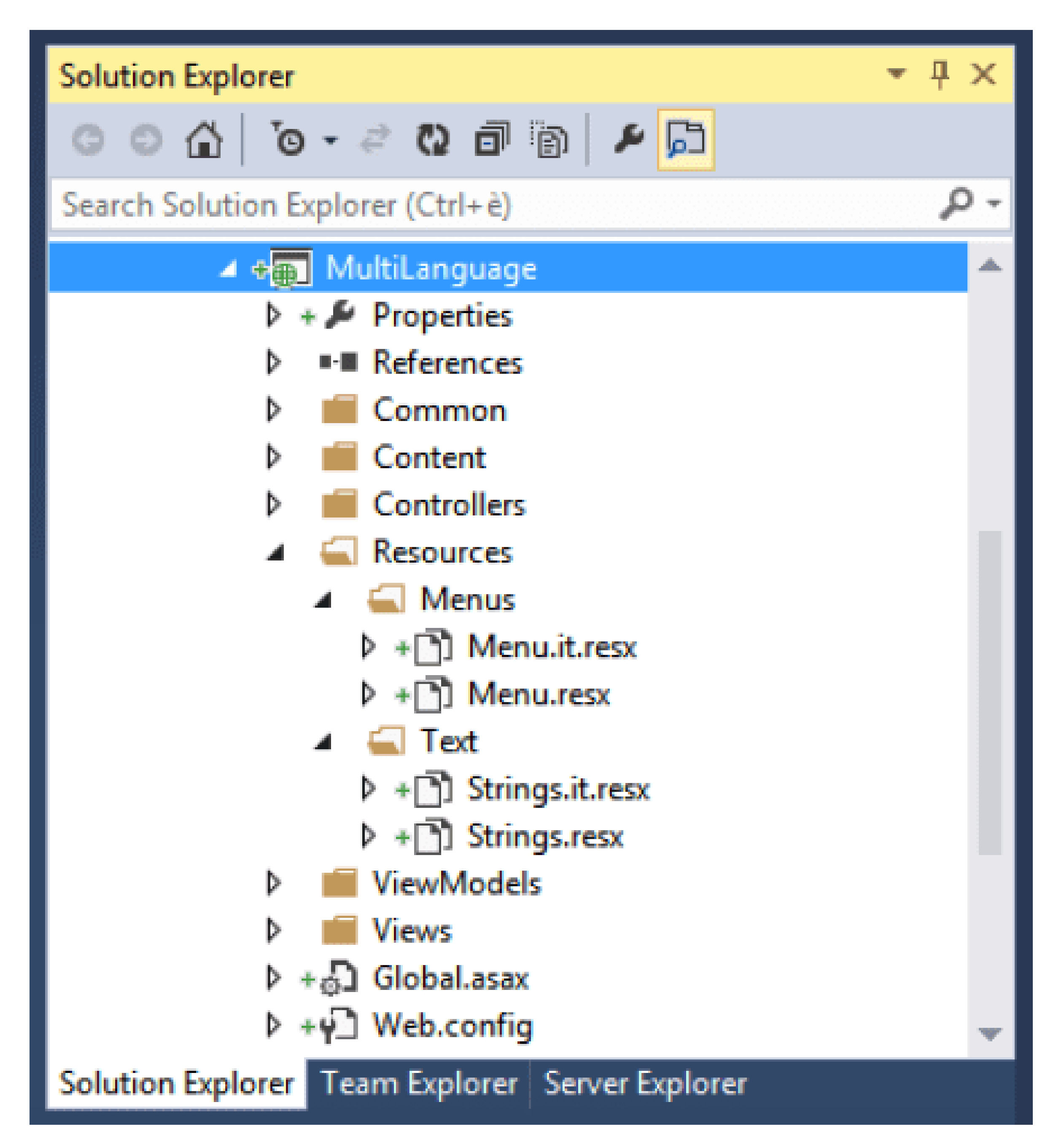


图 5-4　自定义 Resources 文件夹

```
@using MultiLanguage.Resources.Text;
...
<p>@Strings.OurServices</p>
```

上面的表达式保证了能够检索和显示语言中性值或本地值。资源管理器会选出适用于当前区域性的程序集资源。

所有使用默认语言的 RESX 文件都会被编译到与应用程序相同的程序集中。这就是文件名称不包含区域性引用的情况，例如 errors.resx、strings.resx、menus.resx 等。区域性特定的资源在单独的程序集中编译，一个区域一个。将文本本地化到一个区域只要通过添加一个新的 RESX 文件，并根据下面所示的模式命名即可：

```
filename.XX.resx
```

在这里，XX 代表区域的首两个字母；例如，it、fr、en、es、de 等。那些本地化文件是原始(非特定区域性)文件的副本，并仅会将文本转译成对应语言。

小贴士：

我还是建议你考虑将资源甚至是默认资源保存在它们自己的程序集中。你只需要创建一个新的类库项目，将 Resources 文件夹拖放进去(包括本地化的版本)，并从主应用程序引用这个库。

在 ASP.NET MVC 中，在项目内部组织文件的另一个有效方法是将所有资源分组到默认的 Content 文件夹中。在我当前的项目中，Content 中有 Scripts、Styles、Images 和 Text 子文件夹。Text 子文件夹包含 RESX 文件，而 RESX 文件被限制为字符串。

2. 可本地化的文件

在 ASP.NET MVC 中，普遍使用 Url.Content 方法来引用内容文件，比如图片或样式表。这种方法的主要好处是它能显式地将相对路径转换为绝对路径。具体来说，该方法可以解析波浪号(~)运算符。当运用于 URL 时，波浪号运算符能够指示根路径，不论根路径是什么，或你的应用程序是否被部署为根应用程序或子应用程序。因此，我总是建议通过 Url.Content 引用外部文件。

如果此方法可以处理本地化工作岂不是也很好？比如说你请求路径 welcome.jpg，如果当前区域性是 IT(指国家，不是部门)，则返回 welcome.it.jpg。这一方法本身不能改动；但是，如这里所示，编写一个扩展方法从而扩展 Url.Content 语法却很容易：

```
public static class UrlExtensions
{
    public static String Content(this UrlHelper helper,
                                 String contentPath, Boolean localizable=false)
    {
        var url = contentPath;
        if (localizable)
            url = GetLocalizedUrl(helper, url);
        return helper.Content(url);
    }

    public static String GetLocalizedUrl(UrlHelper helper, String resourceUrl)
    {
        var cultureExt = String.Format("{0}{1}",
            Thread.CurrentThread.CurrentUICulture.TwoLetterISOLanguageName,
            Path.GetExtension(resourceUrl));
        var url = Path.ChangeExtension(resourceUrl, cultureExt);

        // Check if localized URL exists, and return file.XX.jpg (or whatever)
        return VirtualFileExists(helper, url) ? url : resourceUrl;
    }

    public static Boolean VirtualFileExists(UrlHelper helper, String url)
    {
        var fullVirtualPath = helper.Content(url);
        var physicalPath = helper.RequestContext. HttpContext.Server.MapPath
            (fullVirtualPath);
        return File.Exists(physicalPath);
    }
}
```

基本上来说，新的 Content 方法是对 Url.Content 的一个简单封装。该方法会检验是否存在本地化的资源。如果确实存在，该方法即返回本地化的 URL；否则，它会返回原始的 URL。下面是如何使用该方法的一个示例：

```
@using BookSamples.Components.Localization;
...
<link href="@Url.Content("~/content/site.css", localizable:true)" rel=
    "stylesheet" type="text/css" />
```

请记住你引用的资源类型(图片、样式表或脚本)是不相关的；Content 方法只是处理 URL，并在条件适用的时候更改扩展名。

3. 引用嵌入文件

当你在一个程序集的资源部分嵌入文件(脚本、CSS 或图片)时，你需要做一些额外的工作来检索它。然而从积极的一面看，你不需要分别部署文件；部署程序集就行了。

在程序集中嵌入资源有两种方式。可以在 RESX 文件中添加资源文件。打开 RESX 文件，使用设计器界面提取出现有的图片或创建一个新的图片。图片是作为位图对象存储在程序集资源中的。可以通过使用 ResourceManager 类或 HttpContext 类上的 GetGlobalResourceObject 方法来检索该信息(后者只适用于 RESX 文件放在老式的 App_GlobalResources 文件夹中)。在这两种情况下，都不能将程序集中的内容转换成可以附加到 HTML 标记的 URL。一种可能的情况是创建一个特设的控制器方法，如下所示：

```
public Object Image(String name)
{
    var bits = (Bitmap) HttpContext.GetGlobalResourceObject("AllResources",
                  name);
    Response.ContentType = "image/jpeg";
    bits.Save(Response.OutputStream, ImageFormat.Jpeg);
    return bits;
}
```

<img>标记看起来如下：

```
<img alt="" src="/home/image?name=trees" />
```

一个更好的方法是将资源作为一个单独的嵌入资源添加到程序集。图 5-5 显示了如何用一个 JPEG 图片来实现这一点。将图片添加到你的项目，然后将生成操作更改为 Embedded Resource。

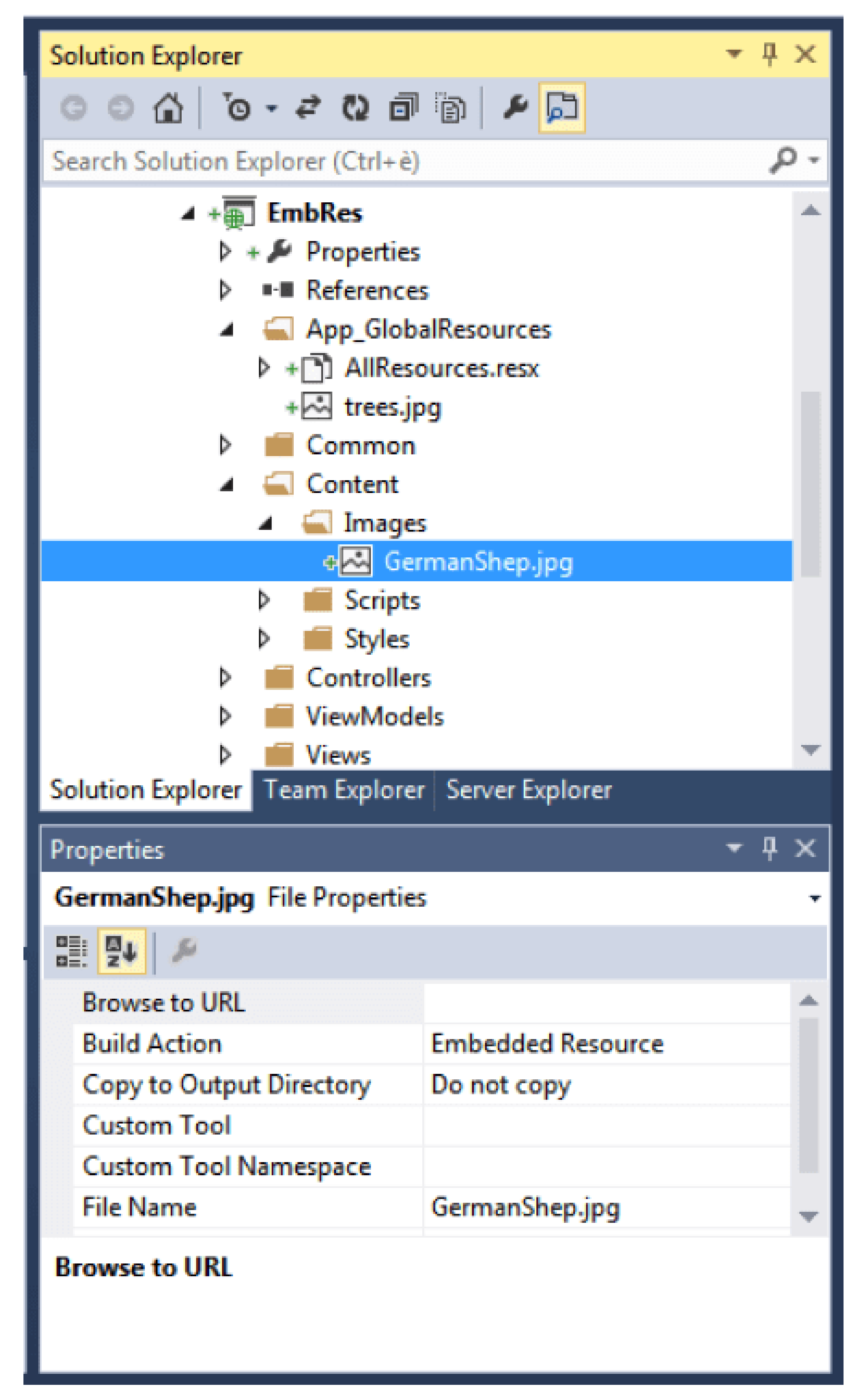


图 5-5　将图片作为嵌入资源添加

接下来更重要的是，把你的程序集修饰成一个包含嵌入资源的程序集。在 Properties 项目文件夹的 AssemblyInfo 文件中添加以下内容：

```
[assembly: WebResource("EmbRes.Content.Images.GermanShep.jpg",
    "image/jpg")]
```

路径就是资源的完全限定名称。如果要检索资源，只需要在 Razor 中这样做：

```
@{
    var p = new Page();
    var url = p.ClientScript.GetWebResourceUrl(typeof(MvcApplication),
                                        "EmbRes.Content.GermanShep.jpg");
}
...
<img alt="" src="@url" />
```

另一个需要考虑的问题是 GetWebResourceUrl 的第一个参数。将其设置为应用程序的类型就好。图 5-6 展现了所显示的图片。

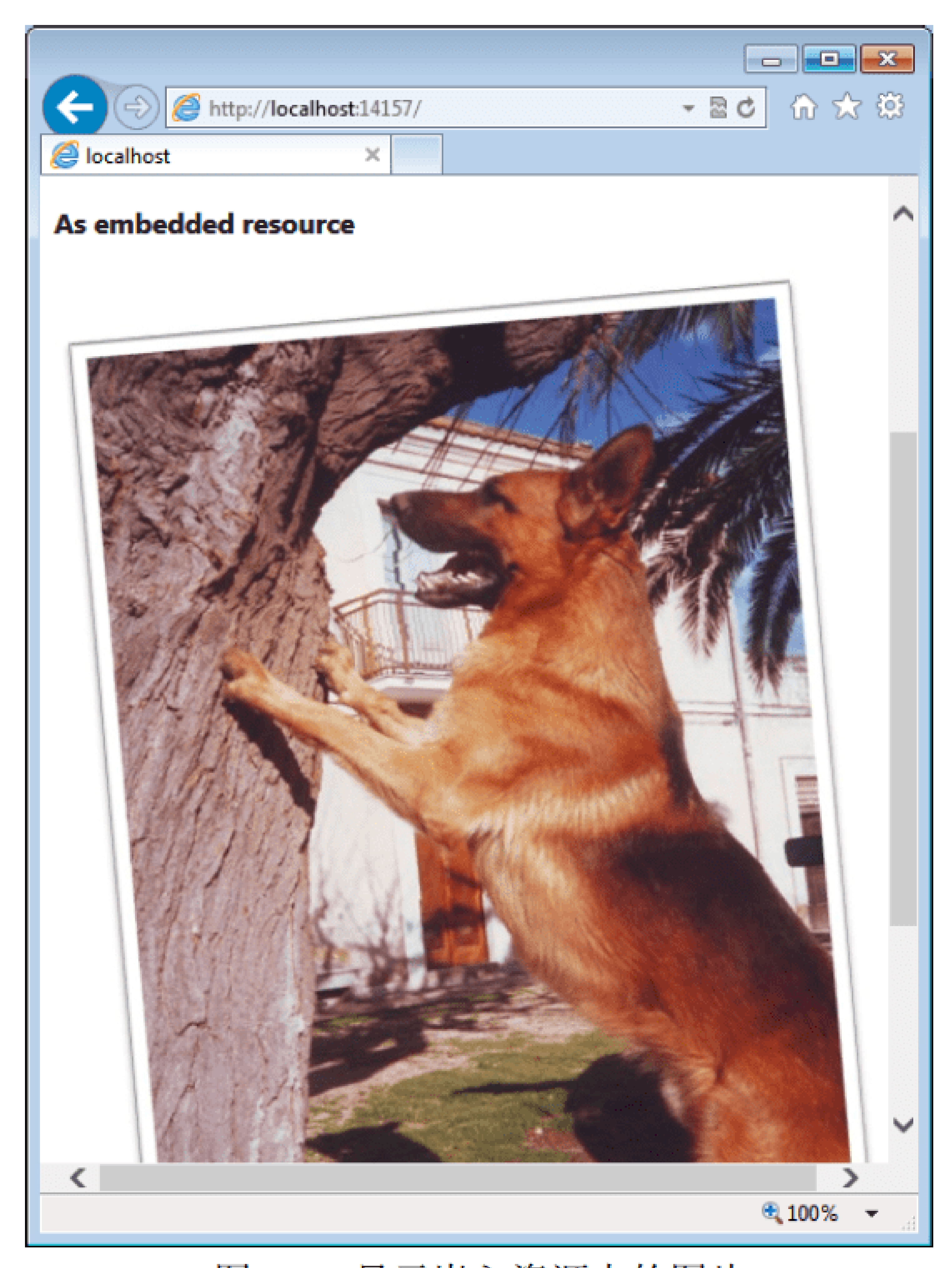


图 5-6　显示嵌入资源中的图片

4. 可本地化的视图

视图是应用程序中可能需要适应当前区域设置的另外一个部分。在 ASP.NET MVC 中，你要从操作方法的内部调出视图。另外，每一个从控制器调用的视图可以包括可能也需要进行本地化的部分视图。这意味着你需要在两个层面添加本地化功能，使用操作方法和扩展方法，扩展方法通常用于链接部分视图。

向操作方法添加本地化逻辑的最好方法是通过操作筛选器来实现。在结束时，你需要做的只是通过算法确定视图的名称并调用它。操作筛选器使代码保持整洁，并将本地化逻辑移到可以单独进行管理的其他地方。可以在第 8 章读到更多有关操作筛选器的信息。现在，让我们把注意力集中到 HTML 扩展上，以调用本地化的部分视图。

在 ASP.NET MVC 中，当你只想呈现视图的时候要使用 Html.RenderPartial；当你想要取回 HTML 标记并自己编写到流时，要使用 Html.Partial。这里添加本地化的逻辑与我们早先用于资源文件的过程几乎相同。下面是扩展 Partial 的一个新的 HTML 扩展方法。

```
public static class PartialExtensions
{
```

```
public static MvcHtmlString Partial(this HtmlHelper htmlHelper,
        String partialViewName, Object model, ViewDataDictionary viewData,
       Boolean localizable = false)
{
   // Attempt to get a localized view name
   var viewName = partialViewName;
   if (localizable)
      viewName = GetLocalizedViewName(htmlHelper, viewName);

   // Call the system Partial method
   return System.Web.Mvc.Html.PartialExtensions.Partial(htmlHelper,
           viewName, model, viewData);
}

public static MvcHtmlString Partial(this HtmlHelper htmlHelper, String
           partialViewName,Boolean localizable=false)
{
   // Attempt to get a localized view name
   var viewName = partialViewName;
   if (localizable)
      viewName = GetLocalizedViewName(htmlHelper, viewName);

   // Call the system Partial method
   return htmlHelper.Partial(viewName, null, htmlHelper.ViewData);
}

public static String GetLocalizedViewName(HtmlHelper htmlHelper,
     String partialViewName)
{
   var urlHelper = new UrlHelper(htmlHelper.ViewContext.RequestContext);
   return UrlExtensions.GetLocalizedUrl(urlHelper, partialViewName);
}
}
```

代码的结构是相当直观的。新方法会根据既定的约定来检查是否存在本地化的视图。如果确实存在，该方法会进一步调用带有本地化名称的原始 Partial 方法；否则，一切照约定进行。额外的步骤可以通过可本地化的 Boolean 参数得以控制：

```
@using BookSamples.Components.Localization;
...
@Html.Partial("_aboutdetails", localizable:true)
```

用于 RenderPartial 的代码几乎相同，可以在下载的源代码中找到。

5.3.2　处理可本地化的应用程序

到目前为止，你已经看到了个别必须本地化的资源是如何处理的。然而，将应用程序本地化不仅仅需要本地化单个资源的集合。尤其是，你应该有一个清晰的概念，打算通过本地化应用程序来达成什么目的。希望应用程序能够支持多种语言，但要在安装时配置每种语言吗？或者，你希望用户能够在预定义的语言之间进行切换吗？又或者，你最终要的是一个自适应的应用程序吗？我们来检视一下这三种情形。

1. 自适应的应用程序

自适应的应用程序是基于用户提供的信息，以某种方式确定要使用的区域性的应用程序。这样的应用程序支持大量的预定义区域性，并且当检测到区域设置不匹配任何所支持的区域性时即退回到非特定的区域性。非特定区域性就是应用程序的本机默认语言。非特定区域性资源是指那些名称中不包含任何区域性的 ID。

自适应应用程序的最典型特征是该应用程序如何确定要使用的区域性。通常有几种方式。最常见的方式是要让应用程序读取随浏览器所发送的各个请求的可接受语言列表。另一种方式是基于地理位置，应用程序的服务器端部分以某种方式获取当前用户(IP 地址或客户端地址)的位置信息并选择相应的区域性。第一种方法直接由 ASP.NET 运行时提供；第二种方法则需要你进行一些额外的工作，并可能使用一些额外的框架。让我们来检视第一种方法。

在 ASP.NET 中，可以使用 Culture 和 UICulture 属性来获取和设置当前区域性。这是在每个请求的基础上——例如，从各控制器类的构造函数内部，或一些控制器基类的构造函数中。Culture 属性支配所有应用程序范围的设置，如日期、货币和数字。UICulture 属性控制用于加载资源的语言。这些字符串属性可以被 ASPX 和 Razor 视图引擎中的视图类公开获取。

请注意这两个属性默认为空字符串，这意味着默认区域性就是设置在 Web 服务器上的区域性。如果区域性属性设置为自动，则由浏览器通过 Accept-Languages 请求标头发送的首选语言会被选出。要让视图自动本地化(如果用于那个区域性的资源可用)，必须向 web.config 文件添加下列行：

```
<system.web>
  ...
  <globalization culture="auto" uiCulture="auto" />
</system.web>
```

要让自适应的本地化应用程序启动和运行，这就是你唯一需要做的(除了使用本地化的资源之外)。

2. 多语言应用程序

另一种可能的情形是，当你有一个部署了多个本地化程序集的多语言应用程序，却被配置为只能使用一个资源集。同样，在这种情况下，你不用在任何位置编写特设的代码，只需要在 web.config 文件的全球化部分写入正确的信息：

```
<system.web>
  ...
  <globalization culture="it" uiCulture="it" />
</system.web>
```

当然，此外你要使区域性特定的资源可用，以便它们可以通过 ASP.NET 框架被自动调用。

3. 以编程方式更改区域性

但是，大多数情况下，你只想能够以编程方式设置区域性，以及在用户通过单击图标或使用区域性特定的 URL 切换到不同的区域性时能够动态更改区域性。

当你要以编程方式更改区域性时，你需要满足两个关键要求。首先，定义要用于检索区域性设置的策略。该策略可以是从某些数据库表或 ASP.NET 缓存中所读取的值。也可以是从 URL 中所检索的值。最后，它也可以是你通过地理位置获取的参数——即通过查看用户连接的 IP 地址。在任何情况下，某一时刻你总会知道标识要设置区域性的神奇字符串。那么你如何应用它呢？

下面的代码显示了与要使用的区域性有关的需要向 ASP.NET 运行时所指示的内容：

```
var culture = "..."; // i.e., it-IT
var cultureInfo = CultureInfo.CreateSpecificCulture(culture);
Thread.CurrentThread.CurrentCulture = cultureInfo;
Thread.CurrentThread.CurrentUICulture = cultureInfo;
```

选取 ASP.NET 当前线程，并设置 CurrentCulture 和 CurrentUICulture 属性。请注意两个区域性属性不一定具有相同的值。例如，可以根据浏览器的配置切换文本语言和消息，同时保留全球化设置(如日期和货币)常数。

必须为每一个请求设置区域性，因为每个请求都在它自己的线程上运行。在 ASP.NET MVC 中，可以用多种方式实现这一点。例如，可以将前面的代码嵌入到一个控制器基类。这将强制从给定的基类派生所有的控制器。如果觉得这不可接受，或者就是想采用另一个路由，则可以借助于一个自定义操作调用程序或全局操作筛选器(第 8 章会介绍调用程序和操作筛选器的更多信息)。在这两种情况下，可以一次性编写代码，并一步式将其附加到所有的控制器。以下是使用全局筛选器进行本地化的代码：

```
 [AttributeUsage(AttributeTargets.Class|AttributeTargets.Method,
AllowMultiple=false, Inherited=true)]
```

```
public class CultureAttribute : ActionFilterAttribute
{
    private const String CookieLangEntry = "lang";

    public String Name { get; set; }
    public static String CookieName
    {
        get { return "_LangPref"; }
    }

    public override void OnActionExecuting(ActionExecutingContext
      filterContext)
    {
        var culture = Name;
        if (String.IsNullOrEmpty(culture))
            culture = GetSavedCultureOrDefault
                        (filterContext.RequestContext.HttpContext.Request);

        // Set culture on current thread
        SetCultureOnThread(culture);

        // Proceed as usual
        base.OnActionExecuting(filterContext);
    }

    public static void SavePreferredCulture(HttpResponseBase response,
      String language,
                                    Int32 expireDays=1)
    {
        var cookie = new HttpCookie(CookieName) { Expires =
        DateTime.Now.AddDays(expireDays) };
        cookie.Values[CookieLangEntry] = language;
        response.Cookies.Add(cookie);
    }

    public static String GetSavedCultureOrDefault(HttpRequestBase
        httpRequestBase)
    {
        var culture = "";
        var cookie = httpRequestBase.Cookies[CookieName];
        if (cookie != null)
            culture = cookie.Values[CookieLangEntry];
        return culture;
    }
```

```
    private static void SetCultureOnThread(String language)
    {
        var cultureInfo = CultureInfo.CreateSpecificCulture(language);
        Thread.CurrentThread.CurrentCulture = cultureInfo;
        Thread.CurrentThread.CurrentUICulture = cultureInfo;
    }
}
```

CultureAttribute 类提供了将特定区域性字符串读写到自定义 cookie 的公共静态方法。筛选器重写了 OnActionExecuting 方法，这意味着它可能在任何控制器方法运行之前介入。但是，要实现这一目的，筛选器必须注册为全局筛选器。在实现 OnActionExecuting 的过程中，筛选器会读取之前存储到 cookie 的用户首选的区域性，并将它设置到当前的请求线程。

下面的代码显示了如何把筛选器注册成适用于应用程序中所有控制器方法的全局筛选器：

```
public class MvcApplication : HttpApplication
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
        filters.Add(new CultureAttribute());
    }
    ...
}
```

有了这种基础架构，就可以将链接添加到你的网页(通常是母版页)以便在运行中切换语言：

```
@Html.ActionLink(Menu.Lang_IT, "Set", "Language", new { lang = "it" }, null)
@Html.ActionLink(Menu.Lang_EN, "Set", "Language", new { lang = "en" }, null)
```

要遵循上面的操作，你需要使用一种操作方法。我喜欢将此代码单独放在一个特定的控制器，如下面所示的 LanguageController 类：

```
public class LanguageController : Controller
{
    public void Set(String lang)
    {
        // Set culture to use next
        CultureAttribute.SavePreferredCulture(HttpContext.Response, lang);

        // Return to the calling URL (or go to the site's home page)
        HttpContext.Response.Redirect(HttpContext.Request.UrlReferrer.
           AbsolutePath);
    }
}
```

该操作方法会直接将新选定的语言存储到你选择的存储区中——在本示例中是一个自定义 cookie——并重定向。图 5-7 显示了一个可以以多种语言显示其内容的示例页。

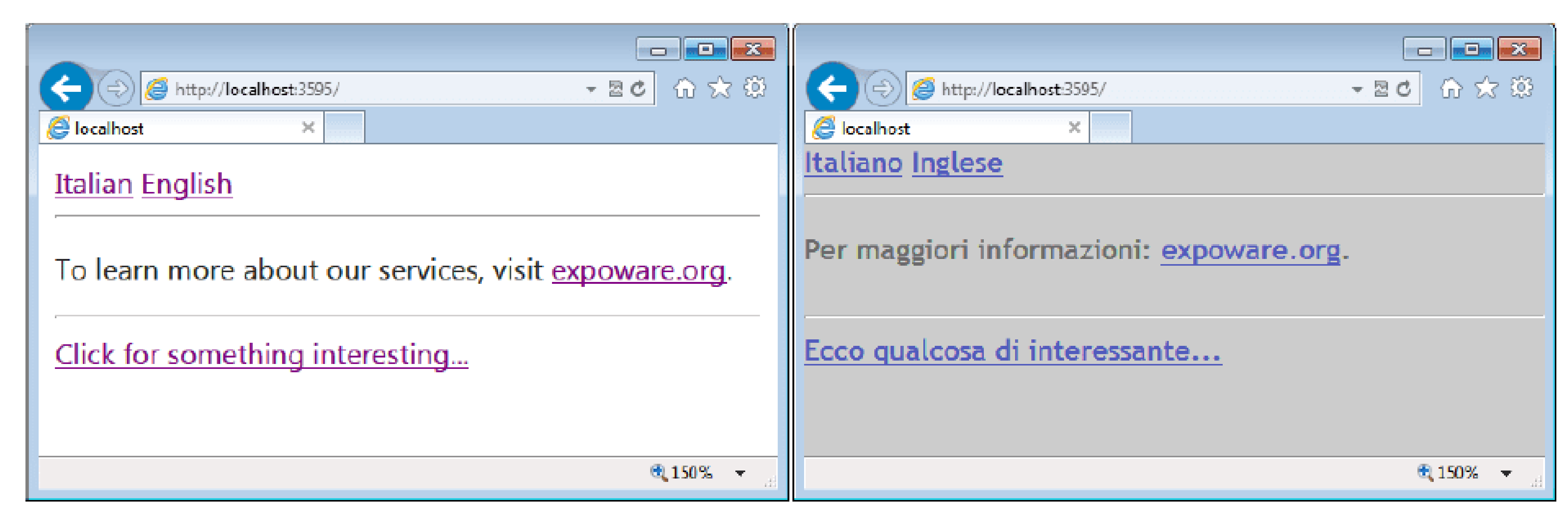


图 5-7　用户可以在运行中切换语言的应用程序

无论用于确定所需语言的技术是通过选择、IP 地址还是地理位置，都可以应用这种方法。

注意：

越来越多的网站从用户的连接中核查位置信息，并推荐一种语言和区域性。此功能需要一个查找 IP 地址的 API，并将其映射到一个国家，然后是一个区域。

在数据库中存储本地化资源

在讨论本地化的时候，似乎不可避免地会谈到数据库作为本地化数据的可能存储区。这是一个可选项吗？答案是肯定的。然而，有一些优缺点需要考虑。

首先，使用数据库会增加延迟，即使你不会为要本地化的视图的每一部分进行数据库的调用。相反，最有可能的情况是你读取一串记录并将其缓存很长时间。因此，通过以这种方式使用数据库所表现出的性能影响比人们第一印象中想象的破坏性要小。

在数据库中存储本地化数据需要一个自定义的本地化层，而通过经典的基于 XML 的资源文件的方法不会导致你编写很多额外的代码，并为你提供来自 Visual Studio 设计器的出色支持。

当视图的数量变得庞大(比如数百个)时，资源项的数目至少会增至数以千计。这时，管理它们会变得棘手。你可能会有很多加载到 AppDomain 占用运行时内存的程序集，这将对站点的整体性能造成影响。因此，数据库或许是大部分本地化内容的最好方式。

存储在关系数据库中的数据更易于管理、查询和缓存，且大小不是问题。此外，有了数据库和自定义的本地化层，可以在本地资源的整体检索过程中获得更多的灵活性。事实上，可以让本地化层提供一组字符串——或者，更好的是提供原始数据——然后格式化以满足用户界面的需要。换句话说，自定义的本地化层会将你从维护资源项与用户界面特定片段之间的直接绑定之中解耦出来。

4. 从服务中获取本地化数据

来自移动应用程序的另一种受欢迎的选择是从本地化的服务中获取数据。该服务可以由筹备网站的同一个团队拥有，甚至也可以是一些第三方服务。更重要的是，此方法可以使你以最小的工作量添加新的语言，同时随着时间的推移还可提高已翻译文本的质量，而无须重新部署任何新的内容。

注意：

基于外部服务的方法和使用本地数据库的方法可以封装在一个 ASP.NET 资源提供程序中——一个使用 IResourceProvider 接口的类。

5.4　本章小结

在 ASP.NET MVC 中的应用程序主要是 Web 应用程序。现代 Web 应用程序比几年前具有更多元化的要求。例如，当今的 Web 应用程序必须是对搜索引擎友好的，并且很可能必须支持全面的本地化，以便能够通过利用用户的特定语言和区域性来驱动用户的操作。最后，提供声名狼藉的黄页面错误(即 ASP.NET 的一个默认错误页面)是令人难以接受的；它仍然存在，但其真的很影响一个网站的声誉(未处理的错误一直是一件坏事，但用户在前几年愿意给予的容忍度绝对是过去的事了)。

基于所有这些原因，任何 Web 应用程序的基础架构(以及在这种上下文环境下的 ASP.NET MVC 应用程序基础架构)都需要更为强大和丰富。尤其是，需要更多关注你能够识别的 URL，并设计用于 SEO 和错误处理的 URL。你需要设计视图和控制器以检查当前的区域设置，并自动调整图形和消息。你还需要检测区域性，让用户在你所支持的语言之间进行切换。

本章提供了如何进行 ASP.NET MVC 开发的详尽概述。在第 6 章“应用程序安全性”中，我们将讨论 ASP.NET MVC 应用程序的安全防护。

第 6 章

应用程序安全性

不怕慢，就怕停。

——*孔子*

安全对用户和开发人员来说意味着很多事情。在 Web 上下文中，安全涉及防止在运行的应用程序中注入恶意代码。同样，安全也涉及防止透露私人数据的操作。最后，安全还涉及构建只有通过身份验证和授权的用户方可访问的应用程序(和应用程序中的各个部分)。

应用程序开发人员最常处理的安全方面的内容当然是用户的身份验证和授权。近来，越来越多的网站也开始通过流行的社交服务提供者实行身份验证。尽管对任何应用程序来说都算不上很理想，甚至有时候用作成员的唯一形式几乎是无效的，但社交服务身份验证正在变得越来越受欢迎。正如你会在本章后面所看到的，ASP.NET MVC 通过面向社交服务的应用程序在网站中提供了集成身份验证和授权的配套设施。

6.1 ASP.NET MVC 中的安全性

ASP.NET 提供了一系列的身份验证和授权机制，是将 Internet 信息服务(IIS)、微软.NET Framework 与操作系统的底层安全服务结合起来实现的。如果 IIS 和 ASP.NET 应用程序在集成模式下工作——这是目前 IIS 7 及较新版本最常见的场景——则请求会通过一条单独的管道，其中包括身份验证步骤和一个可选的授权步骤。如果 IIS 和 ASP.NET 运行他们自己的进程，则一些请求可能要在 IIS 入口通过身份验证或授权，而其他请求(比如 ASPX 页面的请求)会与已认证或匿名用户的 IIS 安全令牌一起移交给 ASP.NET。

起初，ASP.NET 支持三种类型的身份验证方法：Windows、Passport 和 Forms。第 4 种可选项是 None，意味着 ASP.NET 甚至不会尝试执行自己的身份验证，而是完全依赖于由 IIS 所执行的身份验证。在这种情况下，匿名用户可以连接，并且通过使用默认的 ASP.NET 账户就可以访问资源。虽然 Passport 身份验证现已过时并不再使用，但其中一些具有启发性的原则和宗旨被保留了下来，并由新兴的安全标准用于更好的服务，如 OAuth 和 OpenID——社

交网络身份验证背后的协议。

6.1.1　身份验证和授权

Windows 身份验证很少应用于实际的互联网应用程序。Windows 身份验证基于微软 Windows 账户和 NTFS ACL 令牌，因此，它会假定客户端是从运行 Windows 的设备连接的。虽然这在内网方案，可能还包括一些外网的方案中是有用且有效的，但 Windows 身份验证在更多情况下是不具有可行性的，因为 Web 应用程序用户需要在应用程序域中拥有 Windows 账户。

表单身份验证是最常用的收集和验证用户凭据的方式；例如使用用户账户数据库进行验证。

1. 在 ASP.NET MVC 中配置身份验证

在 ASP.NET MVC 和 Web Forms 中，是通过根 web.config 文件中的<authentication>部分选择身份验证机制的。下级子目录继承了为应用程序所选的身份验证模式。默认情况下，ASP.NET MVC 应用程序会被配置为使用 Forms 身份验证。下面的代码片段显示了一段来自 ASP.NET MVC 中(我只编辑了登录 URL)自动生成的 web.config 文件的摘录：

```
<authentication mode="Forms">
<forms loginUrl="~/Auth/LogOn" timeout="2880" />
</authentication>
```

以这种方式配置后，应用程序会在每一次用户尝试访问为通过身份验证的用户所保留的 URL 时，把用户重定向到指定的登录 URL。但是，如何标识一个需要身份验证的 ASP.NET MVC URL(比如控制器方法)呢？

2. 限制对操作方法的访问

当要限制对某操作方法的访问时，请使用 Authorize 特性，并确保只有通过身份验证的用户可以执行它。下面是一个示例：

```
[Authorize]
public ActionResult Index()
{
  ...
}
```

如果将 Authorize 特性添加到控制器类，那么控制器上的任何操作方法都将需要进行身份验证。

```
 [Authorize]
```

```
public class HomeController
{
  public ActionResult Index()
  {
     ...
  }
  ...
}
```

Authorize 特性是可继承的。这意味着可以将它添加到你的控制器基类，并确保派生控制器的所有方法都需要进行身份验证。绝对不要将 Authorize 特性用作全局筛选器。事实上，下面的代码会限制对任何资源的访问，包括登录页面：

```
public class MvcApplication : System.Web.HttpApplication
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
       // Don't do this!
       filters.Add(new AuthorizeAttribute());
       ...
    }
}
```

3. 允许匿名调用方

ASP.NET MVC 提供了另一种与安全相关的特性：AllowAnonymous 特性。当应用于方法时，它会指示 ASP.MVC 运行时在调用方未通过身份验证时也让其通过。AllowAnonymous 方法派上用场的情况是，当把 Authorize 应用在类级别时，之后需要启用对一些方法的自由访问，尤其是登录方法。

4. 处理操作方法的授权

Authorize 特性并不局限于身份验证；它还支持基本的授权形式。任何用该特性标记的方法只能由经过身份验证的用户执行。另外，可以用规定的角色将访问限制在一组特定的通过身份验证的用户上。只要通过给该特性添加两个命名参数即可实现这一点，如下所示：

```
[Authorize(Roles="admin", Users="DinoE, FrancescoE")]
public ActionResult Index()
{
  ...
}
```

如果用户未通过身份验证或没有提供所需的用户名称和角色，该特性会阻止对方法的访

问，并将用户重定向到登录 URL。在刚才所示的示例中，只有 DinoE 或 FrancescoE 用户对方法有访问权(如果他们是 Admin 角色的话)。请注意 Roles 和 Users，如果它们被指定了的话，将合并到逻辑 AND 运算中。

5. 授权和输出缓存

如果需要身份验证和(或)授权的方法也配置为支持输出缓存呢？输出缓存——具体来说是 OutputCache 特性——会指示 ASP.NET MVC 不必每次都真正处理请求，但要返回一个先前计算的且依然有效(即没有过期)的缓存响应。输出缓存功能打开后，用户就可能请求一个已在缓存中且受保护的 URL。那么如何实现呢？

ASP.NET MVC 确保了 Authorize 特性优先于输出缓存。尤其是，只有当用户通过了身份验证和授权，输出缓存层才会为处于 Authorize 中的方法返回一个缓存的响应。

6. 隐藏关键的用户界面元素

你可能也希望通过简单地禁用或隐藏那些能够触发受限制操作方法的操作链接和按钮来阻止用户访问受限制的资源。通过检查当前用户和所分配角色的身份验证状态，如果用户没有适当权限的话，就可以关闭关键输入元素的可见性标志。

我喜欢这种做法，但不会将其作为处理角色和权限的唯一解决办法。最后，隐藏用户界面(UI)元素(这比禁用它们要简单有效)还是很不错的，只要你仍然还通过使用编程检查来限制对操作方法的访问。

6.1.2　将身份验证和授权分开

在 ASP.NET MVC 5 之前，Authorize 特性是 ASP.NET MVC 所支持的唯一与安全相关的操作筛选器。它可以处理身份验证和授权，但有时会忽略一些细节。

为了能够执行方法，Authorize 特性首先会要求对用户进行身份验证。接着，如果指定了 User、Role 或两者都被指定了，Authorize 特性就会检查用户是否被授权(通过名称和角色)访问该方法。产生的影响是，该特性最终不是区分用户是否登录，而是区分谁没有权限调用给定的操作方法。事实上，在这两种情况下，尝试调用操作方法都会将用户重定向到登录页面。

在 ASP.NET MVC 5 中已添加了身份验证筛选器，当身份验证失败时，可以选择干预并控制工作流。但在深入探讨身份验证筛选器之前，让我们先看看如何改善 Authorize 特性以区分匿名用户和未经授权的用户。

1. 匿名还是未经授权

让我们创建一个扩展内置授权特性的增强特性类，如下所示：

```
public class AuthorizedOnlyAttribute : AuthorizeAttribute
```

```
{
    public AuthorizedOnlyAttribute()
    {
        View = "error";
        Master = String.Empty;
    }

    public String View { get; set; }
    public String Master { get; set; }

    public override void OnAuthorization(AuthorizationContext filterContext)
    {
        base.OnAuthorization(filterContext);
        CheckIfUserIsAuthenticated(filterContext);
    }

    private void CheckIfUserIsAuthenticated(AuthorizationContext
      filterContext)
    {
        // If Result is null, we're OK: the user is authenticated and authorized.
        if (filterContext.Result == null)
            return;

        // If here, you're getting an HTTP 401 status code
        if (filterContext.HttpContext.User.Identity.IsAuthenticated)
        {
            if (String.IsNullOrEmpty(View))
               return;
            var result = new ViewResult {ViewName = View, MasterName = Master};
            filterContext.Result = result;
        }
    }
}
```

在新的类中，需要重写 OnAuthorization 方法，并运行一些额外的代码来检查是否获得了一条 HTTP 401 消息。如果正是这样，你之后要检查当前用户是否通过了身份验证，并重定向到自己的错误页面(如果有的话)。可以用 View 和 Master 属性来配置带有用户说明的目标错误视图。

实际影响是，如果因用户未登录导致你得到一个 HTTP 401 错误，就会转到登录页面。否则，如果请求因授权权限而失败，用户就会收到一个友好的错误页面。使用新特性也不会变得更简单，这里就证明了这一点。

```
[AuthorizedOnly(Roles="admin", Users="DinoE")]
```

```
public ActionResult Index()
{
  ...
}
```

从有效性来看，这里呈现的解决方案的确可以利用，但从设计的角度来看，可能还不甚理想。

2. 身份验证筛选器

在 ASP.NET MVC 5 中，身份验证筛选器可以是实现 IAuthenticationFilter 接口的任何类，如下所示：

```
public interface IAuthenticationFilter
{
  void OnAuthentication(AuthenticationContext filterContext);
  void OnAuthenticationChallenge(AuthenticationChallengeContext
    filterContext);
}
```

本地实现该接口的框架中唯一的类是 Controller 类。在内部，身份验证筛选器是在传入请求上进行调用的第一组筛选器。OnAuthentication 方法负责实际的身份验证逻辑，并决定验证是否真的应该发生。你在这里执行的任何操作都只是艰巨任务的开始，用户要通过这些操作输入凭据或其他信息。OnAuthenticationChallenge 方法是你在身份验证完成以后将用户重定向到登录页面或任何附加页面的地方。常见的情形是将用户重定向到他可以输入电子邮件地址的页面，比如，该页面随后可以通过 Facebook、LinkedIn 或 Twitter 进行 OAuth 认证。

需要提醒注意的是，直接将用户定向到登录页面并不需要你编写身份验证筛选器；身份验证筛选器不会改变你在应用程序中编写安全性代码的方式。身份验证筛选器可能只会在你需要对身份验证和授权的默认处理进行自定义时才会派上用场。例如，可以创建一个身份验证筛选器，根据运行时条件，比如 URL 中的参数，在运行中决定请求是否需要身份验证。

```
 [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
   Inherited = true)]
public class OptionalAuthenticationAttribute : FilterAttribute,
   IAuthenticationFilter
{
    public void OnAuthentication(AuthenticationContext filterContext)
    {
        var page = filterContext.ActionDescriptor.ActionName;
        if (CheckYourRuntimeCondition())
        {
            filterContext.Result = new HttpUnauthorizedResult();
```

```
                return;
            }
            else
            {
            }
        }

        public void OnAuthenticationChallenge(AuthenticationChallengeContext
filterContext)
        {
        }
    }
```

作为替代方式，可以用 OnAuthenticationChallenge 将用户重定向到另一个他可以提供额外数据或通过另一种安全级别检查的页面。

快速浏览 Windows 身份验证

尽管 Forms 认证是迄今为止用于 ASP.NET 应用程序的最常见的身份验证机制，但仍然存在一些情形你希望选择 Windows 身份验证。通常情况下，在内网方案中，如果应用程序用户也有了可以由 Web 服务器进行身份验证的 Windows 账户时，就可以使用 Windows 身份验证方法。

当使用 Windows 身份验证时，ASP.NET 与 IIS 会同时工作。真正的身份验证是由 IIS 来执行的，它会使用两种身份验证方法中的一种：基本身份验证或集成 Windows 身份验证。在 IIS 验证了用户身份后，它会将安全令牌传递给 ASP.NET。当配置为在 Windows 身份验证模式下运行时，ASP.NET 便不会执行任何进一步的身份验证步骤，只是使用 IIS 令牌来授权对资源的访问。

例如，我们假设你把 Web 服务器配置为使用集成 Windows 身份验证模式，并关闭匿名访问。当用户连接到 ASP.NET 应用程序时会发生什么？如果本地用户的账户不能与 Web 服务器上的或受信任域中的账户相匹配，那么 IIS 就会显示一个对话框，要求用户输入有效的凭据。接着，如果凭据被认定为有效，IIS 会生成一个安全令牌并交付给 ASP.NET。

6.2　实现成员资格系统

要对用户进行身份验证，你需要某种成员资格系统来提供一些管理用户账户的方法。建立成员资格系统意味着编写用于创建新用户以及更新或删除现有用户的软件和用户界面。同时意味着编写用于编辑用户相关信息的软件，比如用户的电子邮件地址、密码和角色。

如何创建一个用户？通常情况是在某些数据存储区添加新记录。每个数据存储区可以有自己的属性集，但核心任务是一样的，且很大程度上由 ASP.NET 本地成员 API 来提取。在 ASP.NET MVC 中，你要构建一个成员资格系统，将 ASP.NET 成员 API 与一个或两个特定账户控制器相结合。一大堆的视图和视图模型类组成了该基础架构。

注意：

用于 ASP.NET MVC 的微软 Visual Studio 工具会生成一个完全支持身份验证和授权的示例应用程序。虽然它十分实用，但你得到的示例代码并不完全是软件的优势典范。在所有我自己的重要应用程序中，我会直接清除自动生成的所有文件并从头开始构建，这是采用接下来介绍的方法达成的。

6.2.1　定义成员资格控制器

最起码，需要有一个获知用户登录和登出的控制器。下面的示例 AuthController 类显示了获得这种控制器的一个可能途径。下面的代码重现了由 ASP.NET MVC 的 Visual Studio 工具所创建的 AccountController 类：

```
public class AuthController : Controller
{
    [HttpGet]
    public ActionResult Logon()
    {
        // Just displays the login view
        return View();
    }

    [HttpPost]
    public ActionResult Logon(LogonViewModel model, String returnUrl)
    {
        // Gets posted credentials and proceeds with
        // actual validation
        ...
    }

    public ActionResult Logoff(String defaultAction="Index",
      String defaultController="Home")
    {
        // Logs out and redirects to the home page
        FormsAuthentication.SignOut();
        return RedirectToAction(defaultAction, defaultController);
    }
}
```

Logon 方法必须分成两部分：一部分重载仅显示登录视图，另一部分处理提交的凭据，并继续进行实际的验证。Logoff 方法会实现从应用程序中登出，并重定向到指定的页面——通常是应用程序的首页。要登出需要使用 ASP.NET 的本地表单身份验证服务。此外，你还可以考虑为 Logoff 添加以下重载：

```
public ActionResult Logoff (String defaultRoute)
{
    FormsAuthentication.SignOut();
    return RedirectToRoute(defaultRoute);
}
```

该方法会接受一个路由名称而非控制器/操作对以识别返回的 URL。

1. 验证用户凭据

图 6-1 显示了一个登录视图的标准 UI。它包含两个文本框：用于填写用户名和密码，以及一个复选框，询问用户是否希望网站记住这些信息。

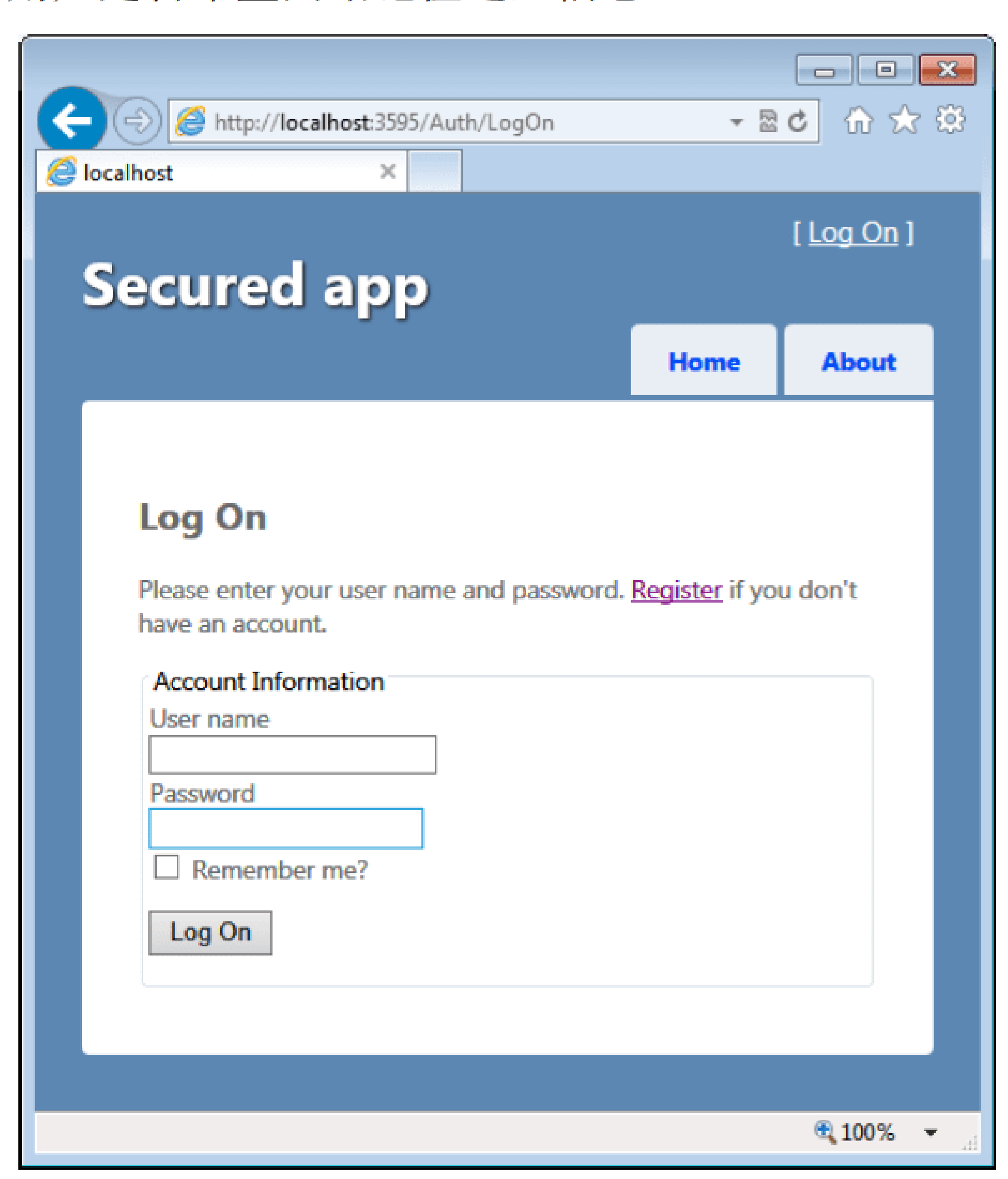


图 6-1　标准登录视图

下面是对从该表单所提交的数据进行处理的控制器方法的可能实现：

```
 [HttpPost]
public ActionResult LogOn(LogonViewModel model, String returnUrl,
              String defaultAction="Index", String defaultController="Home")
{
    var isValidReturnUrl = IsValidReturnUrl(returnUrl);
    if (!ModelState.IsValid)
    {
        ModelState.AddModelError("", "The user name or password provided is
         incorrect.");
        return View(model);
    }

    // Validate and proceed
    if (Membership.ValidateUser(model.UserName, model.Password))
    {
        FormsAuthentication.SetAuthCookie(model.UserName,
         model.RememberMe);
        if (isValidReturnUrl)
        {
            return Redirect(returnUrl);
        }
        else
        {
            return RedirectToAction(defaultAction, defaultController);
        }
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

第 4 章“输入表单”阐释过，输入数据可以在一个易于使用的数据结构中收集并在服务器上处理。这也是你要用 LogonViewModel 类所做的。returnUrl 参数只有在原始请求被重定向到登录视图时才会进行设置，因为用户需要先进行身份验证。在这种情况下，重定向的响应包含位置标头中的新 URL，该 URL 包含一个 returnUrl 查询字符串参数。

验证是通过 ASP.NET 成员 API 发生的。如果验证成功，一个有效的身份验证 cookie 便通过 FormsAuthentication 类创建了。接着，用户会被重定向到最初请求的页面或者首页。

2. 集成成员资格 API

围绕 Membership 静态类，ASP.NET 成员资格 API 使你远离了如何检索和比较凭据及其

他用户信息的细节。Membership 类不直接包含其公开方法的逻辑。实际的逻辑由提供程序组件来提供。

在配置文件中选择成员资格。ASP.NET 附带了几个预定义的提供程序，这些提供程序面向的是微软 SQL Server Express 和活动目录中的 MDF 文件。但是，你最终创建自己的成员资格提供程序也不是不寻常的情况，所以可以重复使用现有的用户数据存储区，并完整控制数据存储区的结构。

定义自定义的成员资格提供程序一点也不难。你只需要从 MembershipProvider 派生一个新类并重写所有的抽象方法。最起码要重写 ValidateUser、GetUser、CreateUser 和 ChangePassword 等几个方法。

```
public class PoorManMembershipProvider : MembershipProvider
{
    public override bool ValidateUser(String username, String password)
    {
        ...
    }

    public override MembershipUser GetUser(String username, Boolean
     userIsOnline)
    {
        ...
    }

    public override MembershipUser CreateUser(String username, String password,
            String email, String passwordQuestion, String passwordAnswer,
            Boolean isApproved, Object providerUserKey,
            out MembershipCreateStatus status)
    {
        ...
    }

    public override Boolean ChangePassword(String username, String oldPassword,
                  String newPassword)
    {
        ...
    }

    // Remainder of the MembershipProvider interface
    ...
}
```

尤其是，在 ValidateUser 的实现中，要提取出用户名称和密码，并将它们与你的数据库进行核对。作为一种安全措施，建议你以哈希格式存储密码。下面是一个将输入密码与存储的哈希密码进行核对的快速演示：

```
public override bool ValidateUser(String username, String password)
{
    // Validate user name
    ...

    // Runs a query against the data store to retrieve the
    // password for the specified user. We're assuming that the
    // retrieved password is hashed.
    var storedPassword = GetStoredPasswordForUser(username);

    // No user found
    if (storedPassword == null)
      return false;

    // Hash the provided password and see if that matches the stored password
    var hashedPassword = Utils.HashPassword(password);
    return hashedPassword == storedPassword;
}
```

默认的成员资格 API 由于其繁琐而备受指责，且明显违反接口隔离原则——也就是流行的 SOLID 首字母缩写中的“I”原则(SOLID 表示单一功能、开闭原则、里氏替换、接口隔离以及依赖反转)。成员资格 API 试图覆盖合理数量的不同情形，但近来对于大部分常见的情形来说，可能过于复杂和丰富。创建自定义的成员资格提供程序有助于解决这个问题，但并不能完全解决，因为它只会构建一个简单的外观。

3. 使用 SimpleMembership API

如果不想从头开始创建自己的成员资格层，可以尝试另一种由 SimpleMembership API 所表示的折中方法选项，它最早用于 ASP.NET Web 页面。

SimpleMembership API 只是在 ASP.NET 成员资格 API 和数据存储区上的封装而已。有了它，就可以使用任何已有的数据存储区，并且它只要求你指明哪些列作为用户名和用户 ID。WebSecurity 类提供了一个简化的 API，用它可以做你的成员资格琐事，其与经典的成员资格 API 的主要区别是从根本上缩短了各个方法的参数列表，并且对存储架构来说，有了更大的自由度。下面的代码显示了如何创建一个新用户：

```
WebSecurity.CreateUserAndAccount(username, password,
new{ FirstName = fname, LastName = lname, Email = email });
```

其结果是，直到 ASP.NET MVC 4 的出现，你才有了两条平行的路由用于成员资格的实现，分别为：围绕 MembershipProvider 类的经典成员资格 API 以及围绕 ExtendedMembershipProvider 类的简单成员资格 API。这两个 API 并不兼容。在 ASP.NET MVC 5 中便努力把这些成员资格实现经验结合起来。但是这需要添加另一个 API——ASP.NET 身份。后面会介绍 ASP.NET 身份。

4. 集成角色 API

ASP.NET 中的角色简化了需要授权的应用程序的实现。角色就是分配给用户的一个逻辑特性。ASP.NET 角色是一个普通字符串，指的是用户在应用程序上下文中所扮演的逻辑角色。从配置方面来说，每个用户可以分配到一个或多个角色。ASP.NET 查找当前用户的角色，并将这些信息绑定到 User 对象。ASP.NET 用一个角色提供程序组件来管理给定用户的角色信息。

角色提供程序是继承自 RoleProvider 类的一个类。角色提供程序的架构与成员资格提供程序的架构没有多大区别，并且有着相同的复杂性问题。要使事情变得简单或封装现有的数据存储区，你可能会希望创建一个自定义的角色提供程序，或切换到简单的角色 API，也就是简单成员资格 API 的角色对应项。

下面的代码片段显示了如何以编程方式将角色和用户相关联。需要使用 Roles 类，作用于具体的角色提供程序。

```
// Create Admin role
Roles.CreateRole("Admin");
Roles.AddUsersToRole("DinoE", "Admin");

// Create Guest role
Roles.CreateRole("Guest");
var guests = new String[2];
guests[0] = "Joe";
guests[1] = "Godzilla";
Roles.AddUsersToRole(guests, "Guest")
```

在运行时，已登录的用户信息可通过 HTTP 上下文的 User 对象获得。下面的代码揭示了如何确定当前用户是否属于某一角色并随后启用特定功能：

```
if (User.IsInRole("Admin"))
{
  // Enable functions specific to the role
  ...
}
```

5. ASP.NET 身份

身份验证的目的是获取与当前用户关联的身份。检索身份以将所提供的凭据与存储在数据库中的记录相匹配。随后，身份系统会基于两个主要区块：用户身份验证管理器和存储管理器。

新的 ASP.NET 身份系统是 Visual Studio 2013 所预示的进行 Web 开发的“One ASP.NET”方法的产物，预计会成为所有 ASP.NET 应用程序处理用户身份验证的首选方法，无论是基于 Web Forms 还是 MVC。ASP.NET 身份的主要目的——Visual Studio 2013 和 ASP.NET MVC 5 中验证用户身份的标准方法——是通过把整个系统分解为两个关键的组件并使用依赖性注入使整个系统易于检验，以统一身份验证的方法。

在 ASP.NET 身份框架中，身份验证管理器采用了 UserManager <TUser>类的形式。该类大体上提供了一个用于用户登录和登出的外观。UserManager 类声明如下：

```
public class UserManager<TUser> : IDisposable where TUser : IUser
{
  :
}
```

TUser 类型指的是要管理的用户的当前描述。IUser 接口包含一个对用户的最低限度的定义，限于 ID 和名称。ASP.NET 身份 API 定义了实现 IUser 接口和添加一些额外属性的 IdentityUser 类型。

```
public class IdentityUser : IUser
{
   public string Id { get; }
   public string UserName { get; set; }
   public string PasswordHash { get; set; }
   public string SecurityStamp { get; set; }
   public ICollection<IdentityUserRole> Roles { get; private set; }
   public ICollection<IdentityUserClaim> Claims { get; private set; }
   public ICollection<IdentityUserLogin> Logins { get; private set; }
}
```

从 Visual Studio 2013 向导处获得的 ASP.NET MVC 5 示例应用程序同样以继承自 IdentityUser 的 ApplicationUser 类为特征，并为进一步的扩展做好了准备。可以通过在下面的类中添加更多的属性，为应用程序要处理的用户轻松地添加额外的配置文件数据：

```
public class ApplicationUser : IdentityUser
{
}
```

用于用户存储的核心类是 UserStore <TUser>。TUser 类型必须是 IdentityUser 或进一步继承的类，如 ApplicationUser。用户存储类实现了 IUserStore 接口，它汇总了在用户存储区允许执行的操作。

```
public interface IUserStore<TUser> : IDisposable where TUser : IUser
{
   Task CreateAsync(TUser user);
   Task DeleteAsync(TUser user);
   Task<TUser> FindByIdAsync(string userId);
   Task<TUser> FindByNameAsync(string userName);
   Task UpdateAsync(TUser user);
}
```

可以看出，用户存储接口看起来很像你围绕数据访问层所构建的典型的存储库接口。UserManager <TUser>和 UserStore <TUser>类存在于不同的名称空间和程序集里。具体来说，UserManager 存在于 Microsoft.AspNet.Identity.Core 中，而 UserStore 在 Microsoft.AspNet.Identity.EntityFramework 中定义并与实际的存储技术有着某种相关性。

整个基础架构在账户控制器类中密切地联系在一起。下面是一个 ASP.NET MVC 账户控制器类的框架，它完全基于 ASP.NET 身份 API：

```
public class AccountController : Controller
{
   public UserManager<ApplicationUser> UserManager { get; set; }
   public AccountController(UserManager<ApplicationUser> manager)
   {
      UserManager = manager;
   }
   public AccountController() : this(
       new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new
   ApplicationDbContext())))
   {
   }
   ...
}
```

该控制器会保留住对身份验证标识管理器的引用。身份验证标识管理器的一个实例是被注入到控制器中。

身份存储区会被注入到标识管理器中，用来验证凭据。但身份存储区需要获知实际的数据源。用户数据通过实体框架代码优先(Entity Framework Code First)进行管理。这意味着你不一定需要创建一个物理的数据库来存储你的用户凭据；相反，可以定义一个 ApplicationUser 类，用基本框架创建一个最合适的数据库来存储这些记录。用户存储区和数据存储区之间的

联系是建立在向导为你创建的 ApplicationDbContext 类中的。

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext() : base("DefaultConnection")
    {
    }
}
```

IdentityDbContext 基类继承自 DbContext，并依赖于实体框架。正是从构造函数中，你会发现该类指的是 web.config 文件中用于读取的表示实际连接字符串的条目。

实体框架代码优先的使用是一重大举措，因为它使数据库的结构成为一个次要重点。你仍然需要它，但可以用代码创建一个基于类的数据库结构。此外，可以使用实体框架代码优先迁移工具修改以前创建的数据库，就像你修改其背后的类一样(参见 http://msdn.microsoft.com/data/jj591621.aspx)

6. 使用 ASP.NET 身份对用户进行验证

ASP.NET 身份也是基于最新的用于.NET(OWIN)身份验证中间件的 Open Web 接口。这意味着进行身份验证(cookies 检查和创建)的典型步骤可以通过抽象的 OWIN 接口来执行，而不是直接在 ASP.NET/IIS 接口中执行。对 OWIN 的支持需要账户控制器具有另一个易用属性，如下所示：

```
private IAuthenticationManager AuthenticationManager
{
   get {
      return HttpContext.GetOwinContext().Authentication;
   }
}
```

IAuthenticationManager 接口是在 Microsoft.Owin.Security 名称空间中定义的。此属性非常关键，因为它需要被注入到包括身份验证相关步骤的所有操作中。让我们来看一个典型的登录方法。

```
private async Task SignInAsync(ApplicationUser user, bool isPersistent)
{
    AuthenticationManager.SignOut(DefaultAuthenticationTypes.ExternalCookie);
    var identity = await UserManager.CreateIdentityAsync(user,
                        DefaultAuthenticationTypes.ApplicationCookie);
    AuthenticationManager.SignIn(
    new AuthenticationProperties() { IsPersistent = isPersistent }, identity);
}
```

要注册一个新用户，需要以下代码：

```
var user = new ApplicationUser() { UserName = model.UserName };
var result = await UserManager.CreateAsync(user, model.Password);
if (result.Succeeded)
{
    await SignInAsync(user, isPersistent: false);
    return RedirectToAction("Index", "Home");
}
```

总而言之，ASP.NET 身份为大部分与身份验证相关的任务提供了一个统一的 API。就个人而言，我喜欢这个 API 的表现力以及将不同形式的身份验证融合在一起的尝试——比如内置和基于 OAuth。另一个很大的加分是与 OWIN 的结合，使其在某种程度上独立于 IIS/ASP.NET 这种特定的运行时。无论你是否在当前项目中使用它，都绝对应该了解一下 ASP.NET 身份。

6.2.2　记住我(Remember-Me)特性与 Ajax

如今，几乎所有的登录视图都标记着“记住我”或“保持登录状态”的复选框，就像 Facebook 那样。通常选择该复选框会使身份验证 cookie 保持更长的时间；更长是多长取决于页面背后的代码。标记仅表明获得更为持久 cookie 的用户偏好，以使他保持连接到站点的时间更长，而不用再次重新键入凭据。

如果该 cookie 过期，在用户下次访问时将被自动重定向到登录页面并请求用户重新输入用户名和密码。这种模式并不新奇，而且开发人员也已经习惯了。但在 Ajax 方案中该模式可能会带来一些问题。

1. 重现问题

想象一下用户单击某处并向服务器发出一个 Ajax 调用的情形。再假设身份验证 cookie 已经过期。随后，服务器会返回一个 HTTP 302 状态码，将用户重定向到登录页面。这就是大家所期望的，不是吗？那么问题在哪里呢？

在 Ajax 方案中，处理请求的是 XMLHttpRequest 对象，而不是浏览器。XMLHttpRequest 会正确处理重定向，并转到登录页。但是，Ajax 调用的原始发出者会重新获得登录页面的标记而非大家预期的数据。其结果是，登录页面很可能会被插入到原始响应预期会插入的任意 DOM 位置，如图 6-2 中所示。

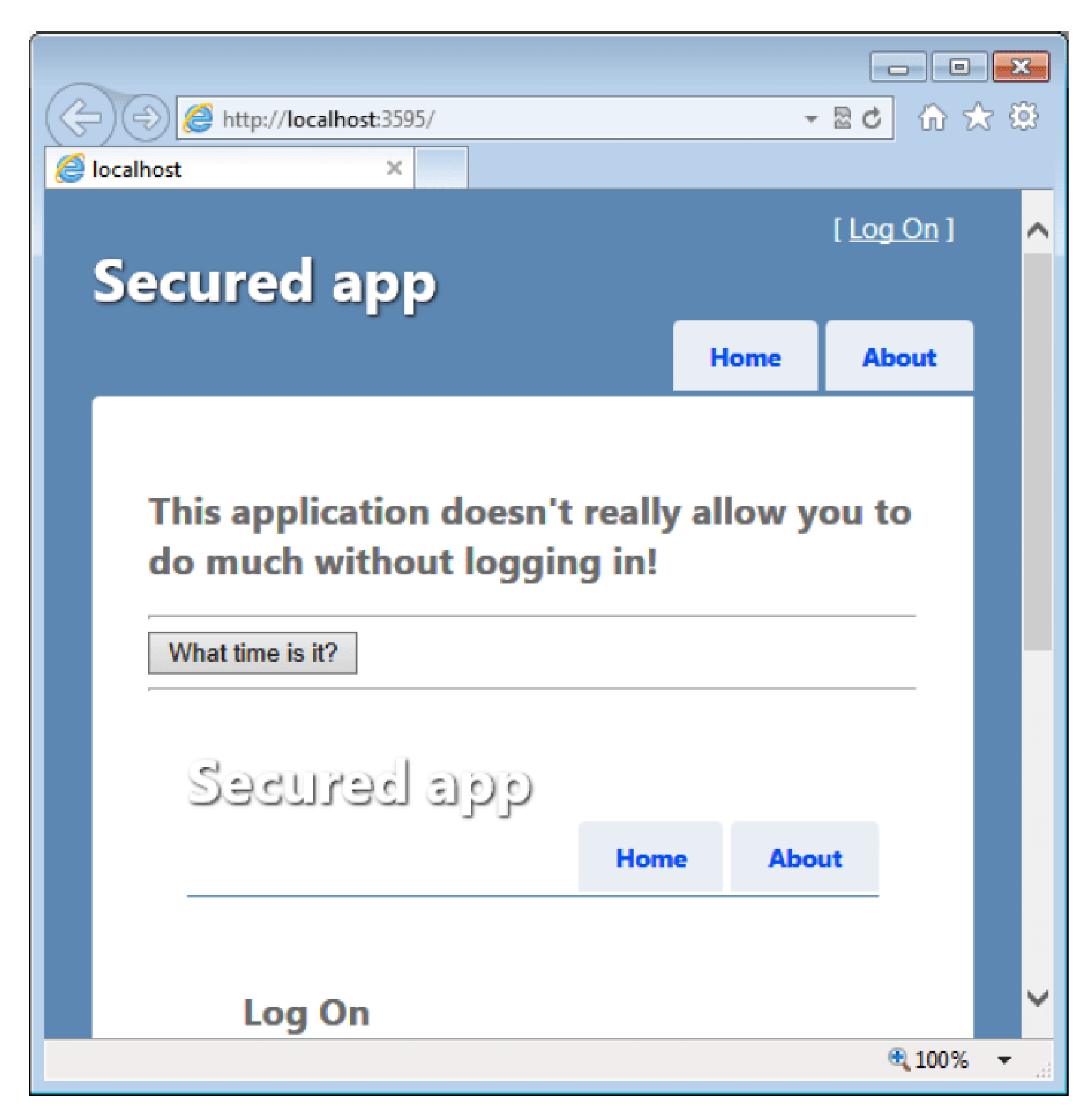


图 6-2　注入登录视图的对受限 URL 的 Ajax 请求

2. 解决问题

要变通解决这一问题，必须在授权阶段拦截该请求并验证它是否是 Ajax 请求。如果是 Ajax 请求且该请求被拒绝，那么你要连接到状态代码，将其更改为 401。之后你所拥有的客户端脚本将其纳入运算，并显示适当的 HTML，如下所示：

```
<script type="text/javascript">
    function failed(xhr, textStatus, errorThrown) {
        if (textStatus == "error") {
            if (xhr.status === 401) {
                $("#divOutput").html("You must be logged in.");
            }
        }
    }
</script>
```

失败的 JavaScript 函数是在表单提交失败或链接请求失败时 Ajax 基础架构使用的回调(如第 4 章中所介绍的)。下面是用于图 6-2 中所示表单的代码：

```
@using (Ajax.BeginForm("Now", "Home",
        new AjaxOptions { UpdateTargetId = "divOutput", OnFailure="failed" }))
{
   <input type="submit" value="What time is it?" />
}
<hr />
<div id="divOutput">
</div>
```

真正的工作是由 Authorize 特性的稍作修订的版本来完成的。总体来说，可以直接扩展 AuthorizedOnly 特性，像下面所示的那样，并在所有有必要限制访问某方法的情况下使用它。用下面这个来替换以前实现的 CheckIfUserIs Authenticated 内部方法：

```
private void CheckIfUserIsAuthenticated(AuthorizationContext filterContext)
{
    // If Result is null, we're OK
    if (filterContext.Result == null)
        return;

    // Is this an Ajax request?
    if (filterContext.HttpContext.Request.IsAjaxRequest())
    {
        // For an Ajax request, just end the request
        filterContext.HttpContext.Response.StatusCode = 401;
        filterContext.HttpContext.Response.End();
    }

    // If here, you're getting an HTTP 401 status code
    if (filterContext.HttpContext.User.Identity.IsAuthenticated)
    {
        var result = new ViewResult {ViewName = View, MasterName = Master};
        filterContext.Result = result;
    }
}
```

有了最后这个更新，AuthorizedOnly 特性就可以成为系统 Authorize 特性的最终替代品。下面是一个可以通过 Ajax 和常规提交进行调用的限制级控制器方法示例。图 6-3 显示了预期的效果。

```
 [AuthorizedOnly]
public ActionResult Now()
{
    ViewBag.Now = DateTime.Now.ToString("hh:mm:ss");
    if (Request.IsAjaxRequest())
```

```
            return PartialView("aNow");
        return View();
    }
    }
```

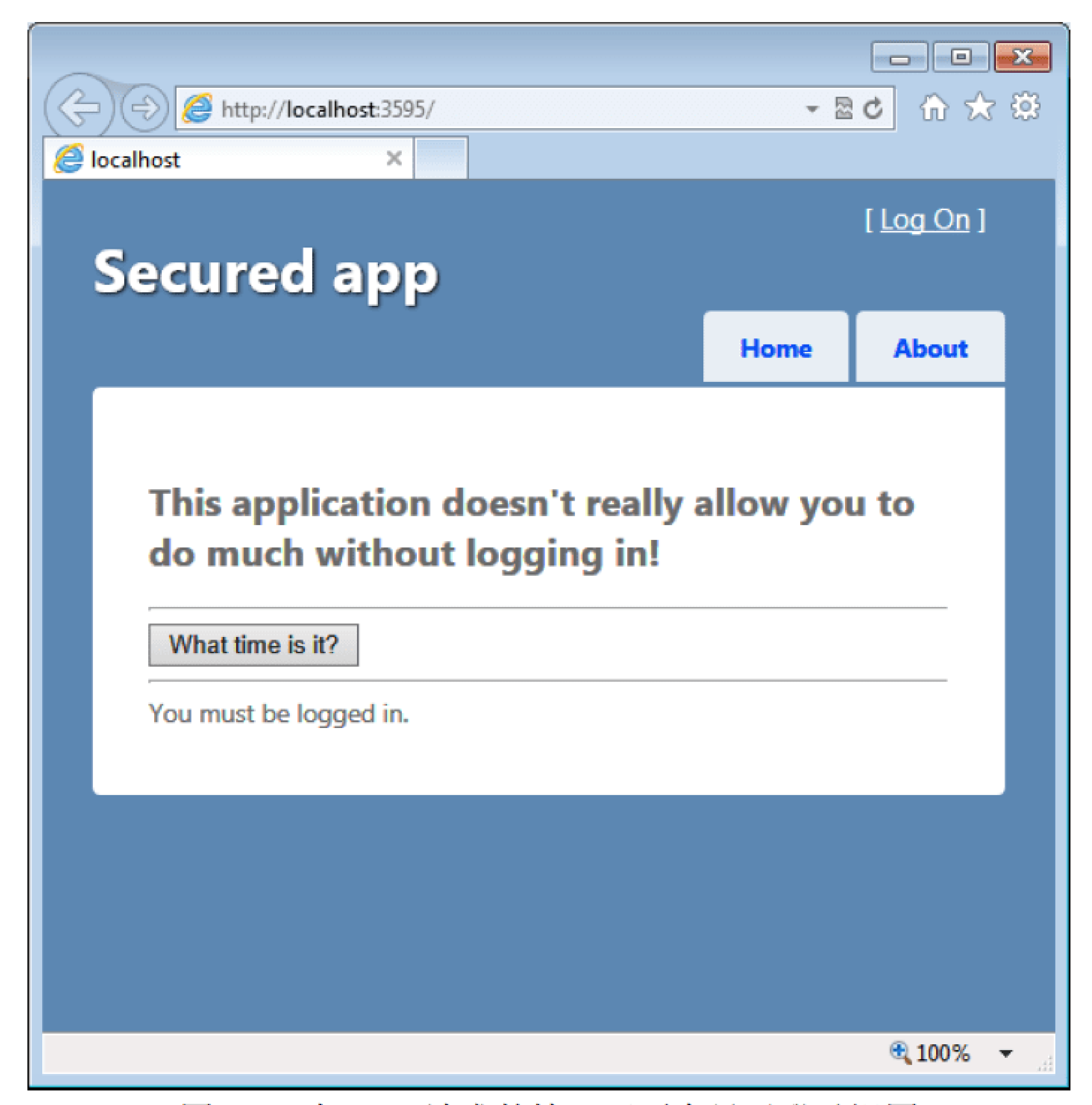


图 6-3　在 Ajax 请求的情况下不会显示登录视图

注意：

“记住我”这个问题只有当你使用 Ajax.BeginForm(或部分呈现)时会体现出来。如果正在对 HTTP 端点进行直接调用并根据你得到的响应更新用户界面，那么一切就都在你的控制之下。此外，如果正调用一个返回 JavaScript 对象标记(JSON)而非 HTML 的端点，那么你会有更多机会去深入理解其响应并做出相应的处理。

6.3　外部身份验证服务

在网站中实现自己的身份验证层绝对是一种选择。然而，近来它变成了只是一种选择而已，甚至可能不是最令用户信服的选择。通过实现你自己的身份验证层，就可以自己负责安全存储密码，并让你的团队负责完成充分管理账户的额外工作。从用户的角度来看，他们感

兴趣的任何新网站似乎都会向列表中添加新的用户名称/密码配对。对于用户来说，忘记密码真的很麻烦。

几年前，微软刚推出的 Passport 方案是一个当用户横跨几个相关网站时为其身份提供便利的早期尝试。有了 Passport，用户只需要登录一次，如果他们成功通过身份验证，便可以在所有相关联的网站上进行自由浏览。

Passport 方案及其相关的 API，现在被官方淘汰了，因为它已经被 OpenID (http://openid.net)所取代。对于依赖流量和大众的网站来说，OpenID 身份验证是一个利益攸关的登录特性，真正有助于吸引和留住更多来到网站的访问者。

6.3.1　OpenID 协议

OpenID 的主要目的是为了使网站访问更简单、快捷，尤其是不要让最终用户感到麻烦。任何支持 OpenID 的网站的访问者都可以使用另一个网站所发出的已有身份令牌来登录。已启用 OpenID 的网站会针对已有的(外部)身份提供程序为用户进行身份验证，并且不需要存储密码和实现成员资格层。

虽然提供了标准，但特定站点的成员资格解决方案仍然是一个选项，很多网站现在还可以让访问者使用从某些提供程序收到的 OpenID 进行身份验证。这一 OpenID 与表单验证或身份验证并不会相互排斥。

图 6-4 提供了一个由支持 OpenID 的网站所采用的身份验证逻辑的全貌图。当用户用一个受支持的 OpenID 点击登录时，网站便会连接到指定的提供程序，并获得该用户的访问令牌。可能会要求该用户输入 OpenID 提供者站点的凭据。已经用提供程序登录的用户也会自动登录到用户感兴趣的网站。在这种情况下，底层发生的这几个重定向绝不会显示中间页面，感觉就与在同一个应用程序的两个页面之间转换一样平滑。

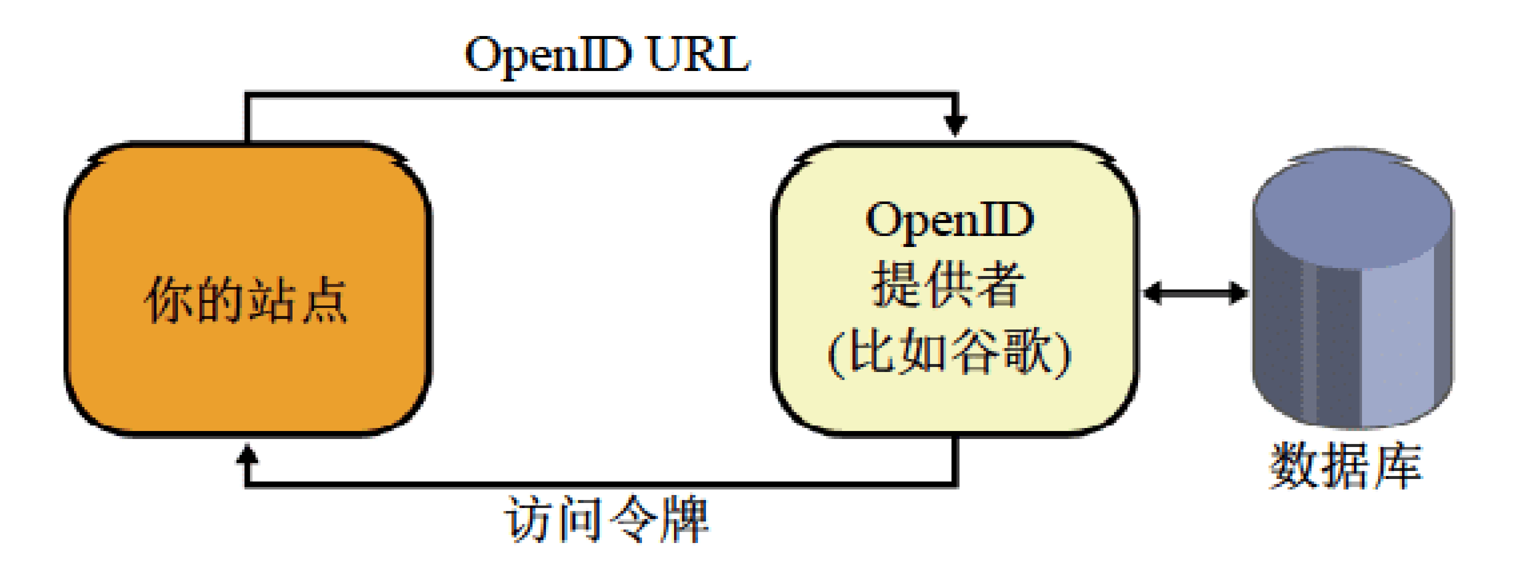


图 6-4　使用 OpenID 的身份验证

你的网站不一定非得是 OpenID 的身份提供者，但却很容易成为如今少数几个可用的 OpenID 提供者所提供的身份令牌的使用者。雅虎、Flickr 和谷歌都是流行的 OpenID 提供者。总体来看，OpenID 与原本的 Windows Live ID 并没有什么不同，差别仅在于它依赖若干服务提供者，且不会强制人们从特定的提供者那里获取另一个账户。通过支持 OpenID，你的用户就可以使用他们已经有的任何凭据登录到你的网站上。用户可以从实质上选择他们从何处

登录。

1. 通过 OpenID 提供者标识用户

有不少程序库会帮助把 OpenID 集成到网站。在.NET 开发人员间流行的一个就是 DotNetOpenAuth(DNOA)，可以从 http://www.dotnetopenauth.net 处获得。图 6-5 显示了一个使用 DNOA 库对用户提供的任何有效 OpenID URL 执行身份验证的示例应用程序。

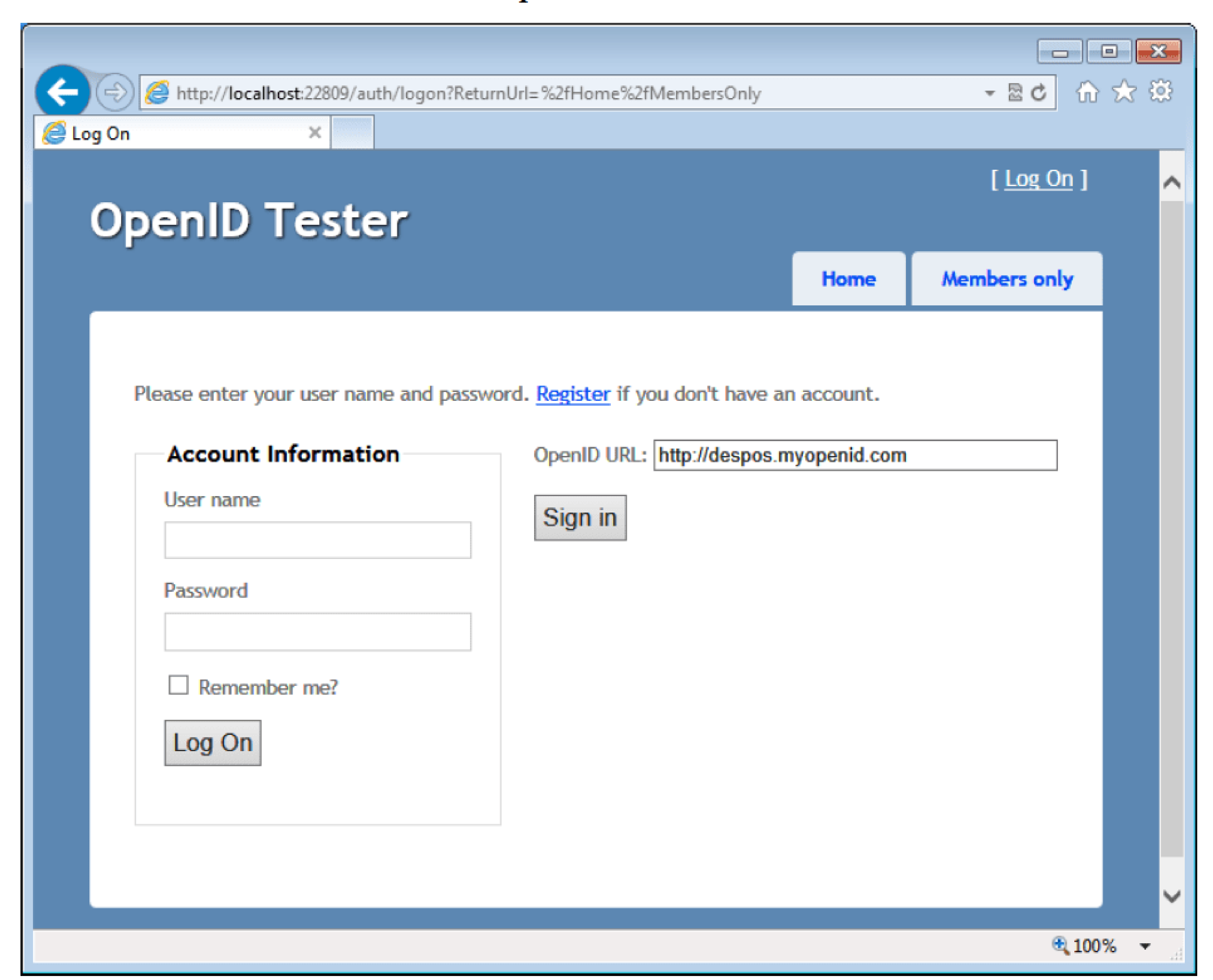


图 6-5　连接到你可能知晓的任意 OpenID 提供者的应用程序示例

作为开发人员，你要抓住的第一条信息就是你准备支持的服务(或多项服务)的 OpenID URL。这个信息对各个用户来说可能相同也可能不同。例如，谷歌和雅虎对不同的用户始终使用相同的 URL。下面分别是谷歌和雅虎的 URL：

- https://www.google.com/accounts/o8/id
- http://yahoo.com

提供程序要在这种情况下从请求的详细信息(比如一个 cookie)中找出用户的账户名称来进行身份验证。如果没有收集到详细信息，或者用户当前未登录到该服务，则该服务会显示登录页面以收集凭据，如果一切都没问题，则该服务会重定向回去(可能是具有进一步访问的cookie)。其他的提供程序，比如 myOpenID (http://www.myopenid.com)，则要求每个用户有一个不同的 URL。特定于用户的 URL 的形式为 http://name.myopenid.com。该服务会从 URL 中确认账户名称，然后按照前文所示的过程执行。

作为开发人员，如果准备支持使用固定 URL 的 OpenID 提供程序，便可以避免出现如图 6-5 中所示的文本框，代之以一个链接或一个按钮。一般情况下，你会希望将用户端的输入降低到最低限度。例如，对于 myOpenID，你只需要输入 URL 的第一部分。下面是用于图 6-5 中所示小表单的 HTML：

```
@using (Html.BeginForm("authenticate", "auth", new { returnUrl = Request.
   QueryString["ReturnUrl"] }))
{
   <label for="openid_identifier">OpenID URL: </label>
   <input id="openid_identifier" name="openid_identifier" size="40" /><br />
   @Html.ValidationMessage("openid_identifier")
   <br />
   <input type="submit" value="Sign in" />
}
```

登录按钮会发送如下所示的一个方法：

```
public ActionResult Authenticate(String returnUrl, [Bind(Prefix=
                                        "openid_identifier")]String url)
{
   // First step: issuing the request and returning here
   var response = RelyingParty.GetResponse();
   if (response == null)
   {
      if (!RelyingParty.IsValid(url))
         return View("Logon");

      try
      {
         return RelyingParty.CreateRequest(url).
          RedirectingResponse.AsActionResult();
      }
      catch (ProtocolException ex)
      {
         ModelState.AddModelError("openid_identifier", ex.Message);
         return View("LogOn");
```

```
            }
        }

        // Second step: redirected here by the provider
        switch (response.Status)
        {
            case AuthenticationStatus.Authenticated:
                FormsAuthentication.SetAuthCookie(response.ClaimedIdentifier,
true);
                return Redirect(returnUrl);

            case AuthenticationStatus.Canceled:
                return View("Logon");

            case AuthenticationStatus.Failed:
                return View("Logon");
        }
        return new EmptyResult();
    }
```

在前面的代码中，该方法归属于包含一个 RelyingParty 属性的控制器类，RelyingParty 属性的定义如下：

```
protected static OpenIdRelyingParty RelyingParty = new OpenIdRelyingParty();
```

这个示例中的控制器由仅提供 RelyingParty 属性的基类派生而来。

OpenIdRelyingParty 类型是在 DNOA 库中定义的。身份验证在同一个控制器方法内分两个阶段开发。CreateRequest 方法准备适当的 HTTP 请求并将它与指定的 OpenID URL 相关联。提供程序接收重定向回到相同 URL 和相同控制器方法的指令。不过，这第二次的响应就不是 null 了，可以创建一个常规的 ASP.NET 身份验证 cookie，它的用户名称就是由 OpenID 提供程序所返回的账户名称。

请注意在这方面你是没有控制权的——提供程序会决定返回什么样的经过身份验证的用户友好名称。例如，出于安全原因，谷歌不会返回有重要含义的友好名称；FriendlyIdentifierForDisplay 属性被设置成通用的 OpenID URL。当你使用 myOpenID 时，它会将响应的 FriendlyIdentifierForDisplay 设置成 URL，并仅仅删除模式信息和尾部斜杠。要完成与 ASP.NET 身份验证基础架构的集成，你需要创建一个常规的身份验证 cookie，让用户名显示在登录区域中。图 6-6 显示了通过 myOpenID 提供程序的身份验证步骤。

图 6-7 显示了用户成功通过 myOpenID 身份验证并返回最初请求站点时所出现的页面。

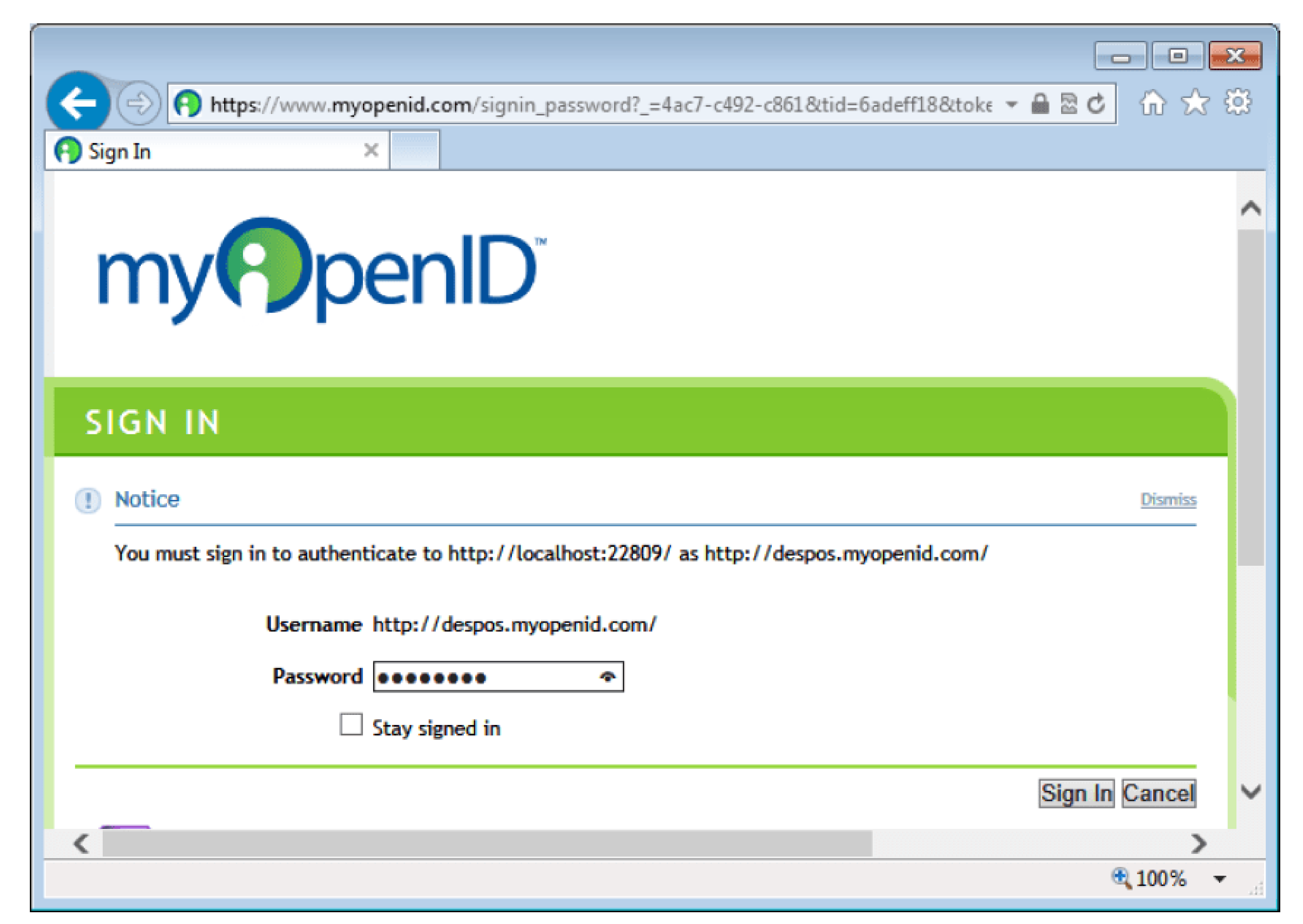


图 6-6　使用 myOpenID 的身份验证阶段

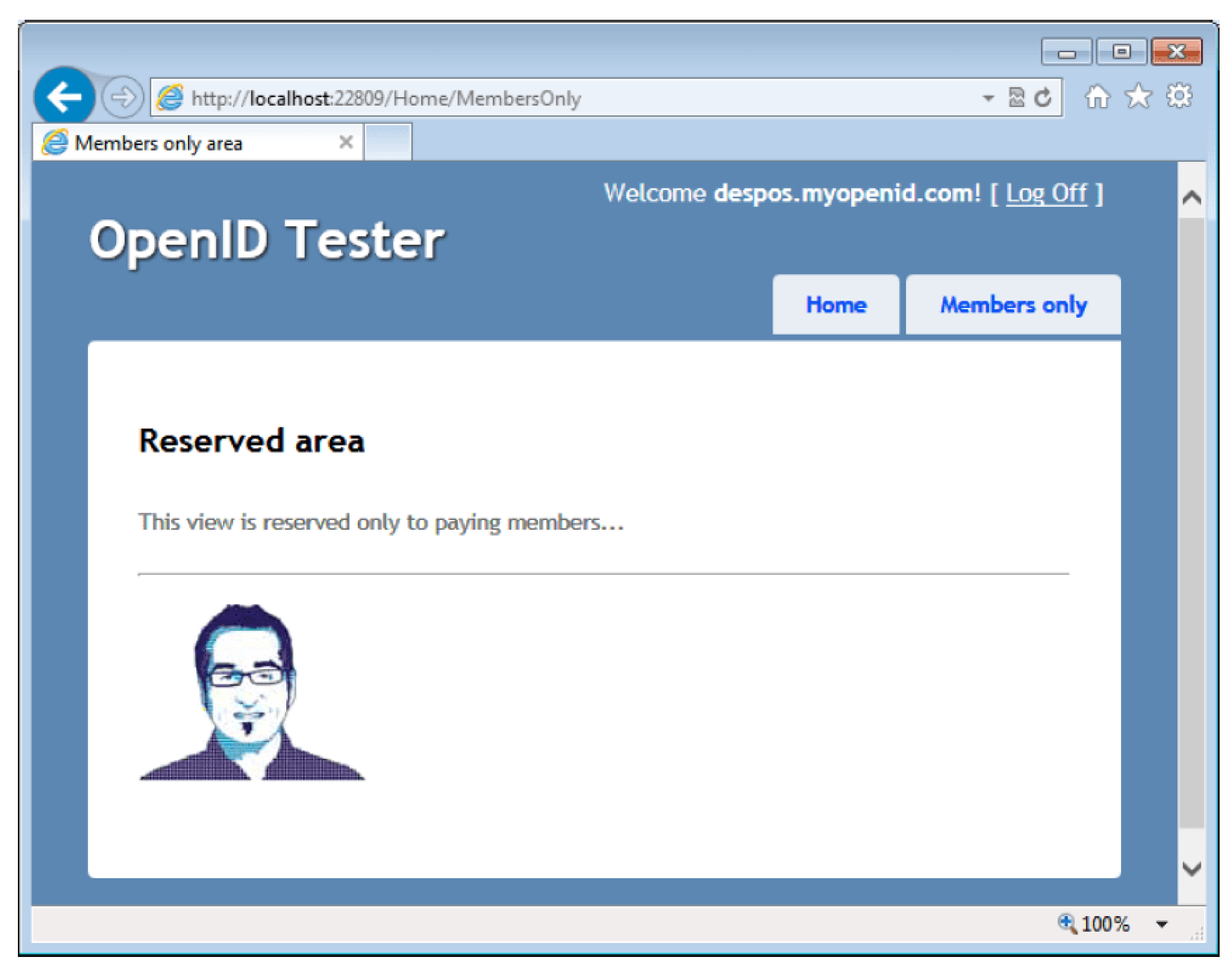


图 6-7　使用 myOpenID 的用户身份验证

用户名称取决于你创建该 cookie 时传递的第一个参数(第二个参数是你是否需要一个持久性 cookie 来保持用户的登录状态)。

```
FormsAuthentication.SetAuthCookie(response.ClaimedIdentifier, true);
```

这里可以自由使用名称。如果仍然保有一份特定于应用程序的昵称列表的话，就可以将声明的标识符映射到昵称并显示这个昵称。或者，可以直接更新 UI 来告知成功登录的用户，而不用显示用户名称。还有另一种选择是创建一个需要自定义的身份验证 cookie，这里你需要使用身份验证票据(cookie 的实际内容)的 UserData 属性来持续存储要显示的友好名称。这种方法的好处是，你既保存了声明的标识符，又保存了你自己合适的友好名称。下面是一个创建自定义身份验证 cookie 的扩展方法：

```
public delegate String UserNameAdapterDelegate(String userName);
public static HttpCookie CreateAuthCookie(this IAuthenticationResponse
response,
                            Boolean persistent = true,
                            UserNameAdapterDelegate fnAdapter = null)
{
    var userName = response.ClaimedIdentifier;
    var userDisplayName = response.FriendlyIdentifierForDisplay;
    if (fnAdapter != null)
        userDisplayName = fnAdapter(userDisplayName);

    return CreateAuthCookie(userName, userDisplayName, persistent);
}

private static HttpCookie CreateAuthCookie(String username,
                                    String userDisplayName,
                                    Boolean persistent)
{
    // Let ASP.NET create a regular authentication cookie
    var cookie = FormsAuthentication.GetAuthCookie(username, persistent);

    // Modify the cookie to add friendly name
    var ticket = FormsAuthentication.Decrypt(cookie.Value);
    var newTicket = new FormsAuthenticationTicket(ticket.Version,
          ticket.Name, ticket.IssueDate, ticket.Expiration, ticket.
           IsPersistent, userDisplayName);
    cookie.Value = FormsAuthentication.Encrypt(newTicket);

    // This modified cookie MUST be re-added to the Response.Cookies collection
    return cookie;
}
```

下面是将此扩展方法放入账户的控制器中的 Authenticated 方法略作修改的版本：

```
switch (response.Status)
{
case AuthenticationStatus.Authenticated:
   var cookie = response.CreateAuthCookie(true, StopAtFirstToken);
   Response.Cookies.Add(cookie);
   if (isValidReturnUrl)
      return Redirect(returnUrl);
return RedirectToAction("Index", "Home");
...
}
```

UserNameAdapterDelegate 委托表示一个函数的模板，可以注入这个函数以确定要显示的友好名称。

```
public String StopAtFirstToken(String name)
{
var tokens = name.Split('.');
return tokens[0];
}
```

最后，需要编辑登录视图来显示 cookie 中 UserData 字段的内容，而不是标准的用户名称。

```
@if(Request.IsAuthenticated) {
   <text>Welcome <strong>@(((FormsIdentity)User.Identity).
      Ticket.UserData)</strong>!
   [ @Html.ActionLink("Log Off", "LogOff", "Auth") ]</text>
}
```

图 6-8 显示了最后的结果。

注意：

当要在身份验证 cookie 中添加更多数据时，内部票据结构的 UserData 属性是首先要考虑的选项。UserData 属性正是为此目的而存在的。不过，你随时都可以创建一个额外的完全自定义的 cookie，或者直接将值添加到身份验证 cookie 中。身份验证 cookie 的名称源于 FormsAuthentication.FormsCookieName 属性的值。



图 6-8　用户已登录，且已显示自定义昵称

2. OpenID 与 OAuth

OpenID 是一个单点登录的方案，因此，它旨在以可能的最简单方式唯一地标识用户。OpenID 不涉及授予用户对服务提供者所管理的资源的访问权。或者，换句话说，OpenID 提供者管理的唯一资源是注册用户的身份。

社交网络的出现——比如 Facebook、Twitter 和 LinkedIn，将经典的单点登录问题置在了不同的聚光灯下。用户不仅希望使用单个(喜爱的)身份登录到多个网站，且无须个个都注册，他们也希望被准予访问自己在网站上的信息和资源，比如帖子、Twitter 文、关注者、朋友、联系人等。

与 OpenID 相比，OAuth(http://oauth.net)是另一个具有附加功能的单点登录方案。充当 OAuth 提供者的网站会作为一个标识提供程序运行，当用户登录时，OAuth 提供程序指定了对资源的权限。提供 OAuth 身份验证的网站会使用特定的 OAuth 协议充当提供程序的客户端。

这样的网站会对用户进行身份验证并获得一个访问令牌，该令牌可进一步用于访问资源(例如，合并 Twitter 文或与它自己的用户界面的联系人)。最后，从用户的角度来看，OAuth 会授予网站(或桌面应用程序)用户对一个账户的受控访问，同时不用泄露登录的详细信息。

最流行的 OAuth 提供者是 Twitter 和 Facebook。

注意：

总体而言，我不认为网站开发人员需要为身份验证问题在使用 OpenID 和 OAuth 之间做选择。你决定需要对哪个外部提供者进行验证(比如 Twitter)，然后结合它所需的 API，无论是 OpenID、OAuth 还是某个专有的 API。如果正在开发一个网站，希望能允许你的用户把他

们的账户(和相关信息)分享给其他网站，那就需要在 OpenID、OAuth 和某专有协议之间作出选择了。在这种情况下，我建议你首先考虑选用 OAuth。

重要提示：

通过社交网络进行身份验证是如今所有预计会有目标消费者的网站都需要的功能。通过社交网络进行的身份验证广受用户的赞赏，因为用户不必费心创建另一个账户和记住另一对凭据那么麻烦。然而与此同时，社交网络也可能屏蔽掉了对网站开发者来说很必要的用户信息片段。

如果要求用户登录，很可能是想要得到他们的电子邮件地址。但通过社交网络可能就不那么容易做到了。为此，以消费者为导向的网站的普遍做法是首先让你选择通过 Twitter 还是 Facebook 进行身份验证，然后让你输入电子邮件地址以及其他个人信息，以完成登录或注册的步骤。

6.3.2　通过社交网络进行身份验证

让我们看看通过 Twitter 和 Facebook 对用户进行身份验证都需要些什么。在接下来的示例中，我们将使用相同的 DNOA 库来封装协议的详细信息。我会将代码限于进行身份验证(本章的例子中 OpenID 和 OAuth 有少量的重叠)，但会调用出那些 OAuth 扩展 OpenID 方案的点。

1. 用 Twitter 注册你的应用程序

OAuth 提供程序的处理需要一些预备步骤。首先你必须用社交网络注册你的应用程序，得到两个字符串：消费者键值和消费者密钥。在对提供程序进一步的编程请求中，你将要嵌入这些字符串。

这一处理过程在 Facebook 和 Twitter 中几乎是相同的。我会在这里讨论 Twitter。整个处理过程从 http://dev.twitter.com 开始。图 6-9 显示了一个现有应用程序的配置页面。

特别是，你要在用户资源上设置应用程序的类型和默认权限。回调 URL 字段值得关注。该 URL 是 Twitter 在成功对某用户进行身份验证之后所返回的。你不太可能希望这个 URL 是固定的。但是，如果应用程序是 Web 应用程序，Twitter 就会要求不要让该字段为空。如果因为你打算为每一个调用指定一个回调 URL 而让这个字段为空的话，那么 Twitter 就会重写设置的应用程序的类型，强制设置为桌面应用程序。实际效果是，对任何用户进行身份验证的尝试都会返回未授权。分配任意一个 URL，包括导致 404 消息的 URL(如本示例所示)都会起到作用。

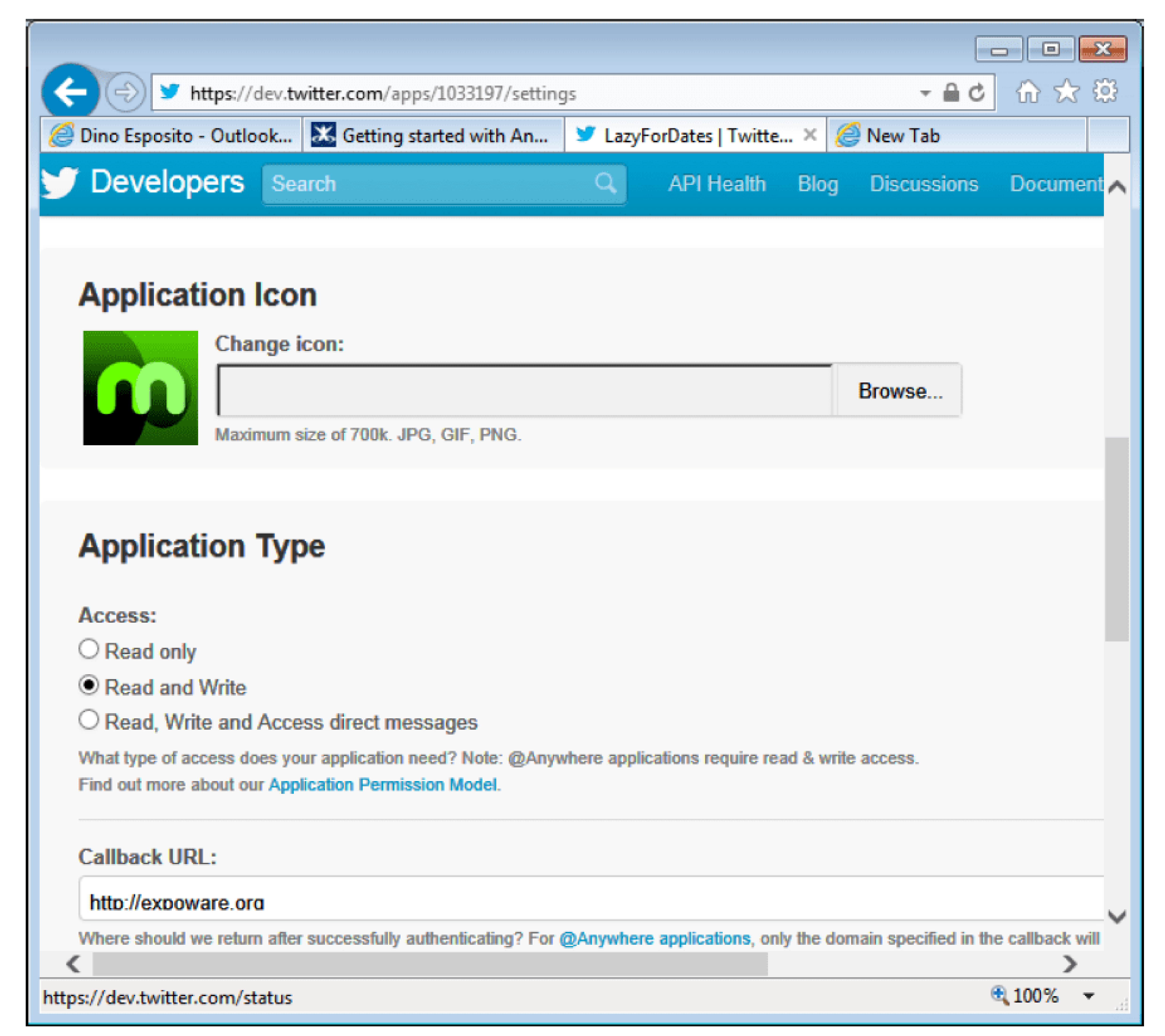


图 6-9　用 Twitter 注册一个应用程序

2. 在 ASP.NET MVC 中启用社交身份验证

在 ASP.NET MVC 4 之前，你都不得不用自己的方式编写在 Twitter 和 Facebook 上进行身份验证的代码。DNOA 库的帮助良多，但它所提供的只是一个低级别的编程接口，这与 OAuth 协议的特点非常接近。在 ASP.NET MVC 中，你会发现一个可用的新类——OAuthWebSecurity 类——它提供了对 OAuth 身份验证的全面支持。尤其是，该类提供了一些可以插入支持 OAuth 的大众社交网络的便利方法。

身份验证方法采用了 RegisterXxxClient 的形式。并提供了支持 Twitter、Facebook、谷歌、领英、雅虎和微软的现成方法。

如果从标准模板启动你的 ASP.NET MVC 项目，就会发现从 global.asax 处调用的如下代码：

```
OAuthWebSecurity.RegisterTwitterClient(
    consumerKey: "...",
```

```
        consumerSecret: "...");

    OAuthWebSecurity.RegisterFacebookClient(
        appId: "...",
        appSecret: "...");
```

确切地说，此代码会被注释掉，并且必须以用于社交应用程序的实际密钥和安全密码在后台完成。然而，即使你不使用任何标准项目模板，也建议看看默认模板中用于社交身份验证的源代码。

只要有了代码的前面两行，就可以从预定义的 ASP.NET MVC 模板中得到如图 6-10 所示的结果。

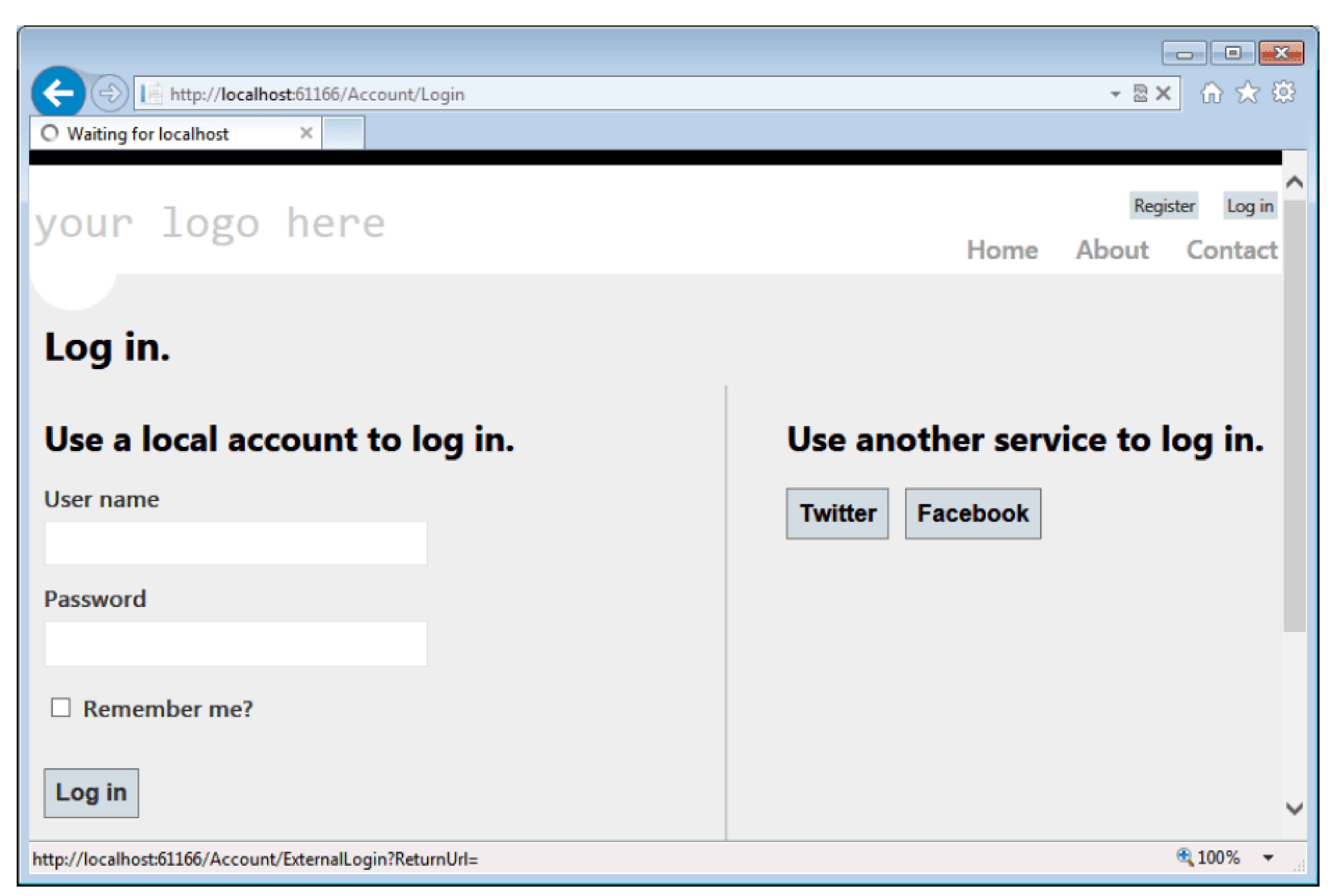


图 6-10 具有双重接口的登录页面示例

如果用户键入用户名和密码并单击 Log In 按钮，一切将正常进行，验证和凭据存储的担子就落在了你的身上。否则，会将用户重定向到 Twitter 或 Facebook 进行身份验证。如果身份验证成功，该用户又会被重定向回来。

整个处理过程使用了 DNOA 库编码；但是，ASP.NET MVC 提供了一些简化编码的封装类，更重要的是，提供了通过社交网络抓取用户信息以及之后将数据保存到本地成员资格系统的框架。

重要提示：

我不知道有多少开发人员实际上从 Visual Studio 标准模板开始构建 ASP.NET MVC 应用程序的。然而，就社交身份验证而言，标准模板中的实现模式绝对值得一看。

3. 启动身份验证处理过程

当用户单击 Twitter(或 Facebook)按钮时，网站最终会调用 Account 控制器上的 ExternalLogin 方法。下面是所涉及的代码：

```
public ActionResult ExternalLogin(String provider, String returnUrl)
{
    return new ExternalLoginResult(provider,
        Url.Action("ExternalLoginCallback",
                   new { ReturnUrl = returnUrl }));
}
```

ExternalLoginResult 类是用于以下代码的封装，它切实做着联系身份验证网关系统的工作：

```
OAuthWebSecurity.RequestAuthentication(Provider, ReturnUrl);
```

ExternalLoginResult 类也是可以在 AccountController.cs 文件中找到的帮助器类。你应该要注意到在项目模板代码中，提供程序的名称是通过查看按钮的名称特性来解决的。

```
<button type="submit"
        name="provider"
        value="@p.AuthenticationClient.ProviderName"
        title="Log in using your @p.DisplayName account">
  @p.DisplayName
</button>
```

在一天结束时，RequestAuthentication 方法会接收到身份验证提供者(Twitter、Facebook 或其他任何受支持的提供者)的名称和返回的 URL。也可以直接从登录控制器方法的调用中提供这一信息。

当请求到达 Twitter 网站时，用户会被重定向到如图 6-11 中所示的授权页面。

如果用户(在这个例子中是我的账号)已经登录到 Twitter，他只会被要求向请求的应用程序授权。否则，他需要首先登录 Twitter，然后授权。接下来，Twitter 会重定向回指定的 URL。

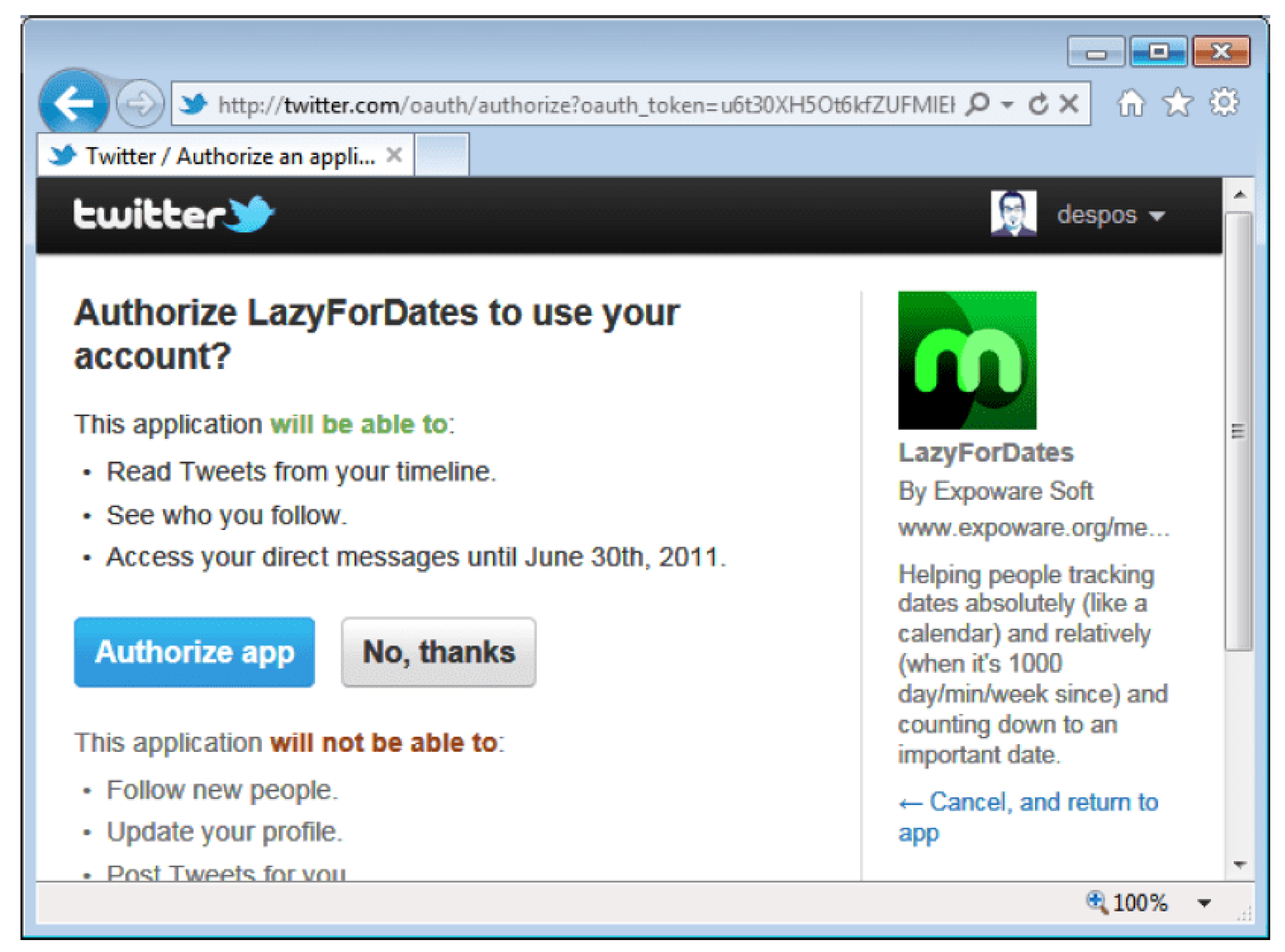


图 6-11　Twitter 上的用户身份验证并授予请求应用程序访问账户信息的权限

4. 处理 Twitter 响应

如果用户输入了 Twitter(或选择的社交网络)能有效识别的凭据，Twitter 网站就会重定向回所提供的返回 URL。另一种方法可以让你重拾通过身份验证的控制，这就是 ExternalLoginCallback。

现在你只知道试图访问你应用程序的用户已经被成功识别为 Twitter 用户。你不知道有关他的任何信息；甚至是用户名称。我很难想到一个需要进行用户身份验证的应用程序可以无关痛痒地忽略用户名或电子邮件地址。身份验证步骤结束时，应用程序只会接收到一个代码，但尚未被授权以编程方式访问 TwitterAPI。为此，在这一阶段收到的代码必须被交换成访问令牌(通常有时间限制以防止滥用)。这是对你在 ExternalLoginCallback 正文中找到的 VerifyAuthentication 方法进行调用的目的。

从 VerifyAuthentication 返回的 AuthenticationResult 对象会带回一些有关用户的信息。你得到的实际信息可能会根据不同的提供者而略有不同；但一般至少会包含用户名称。

5. 从身份验证到成员资格

对用户进行身份验证只是第一步；接着，你需要根据用户名在网站中跟踪用户。在经典的 ASP.NET 成员资格系统中，你首先要显示一个登录表单、验证凭据，然后创建一个完全填充用户名称和其他选择性关键信息的身份验证 cookie。Twitter 和 Facebook 为你省去了设置登

录表单、验证凭据，以及存储和管理像密码这样的账户敏感信息的任务。

不过，差不多所有需要进行用户身份验证的应用程序最起码都需要一个能按名称跟踪各个普通用户的成员资格系统。建立这样一个系统仍然是你自己必须完成的任务。如前所述，ASP.NET MVC 基本模板提供了一个额外的步骤来帮忙解决该问题，在此期间用户会自动获取一次输入其显示名称的机会，然后保存到本地成员资格表。这只有在用户首次登录到指定网站时才需要。换言之，图 6-12 中所示的表单会服务于连接注册和首次登录的目的。

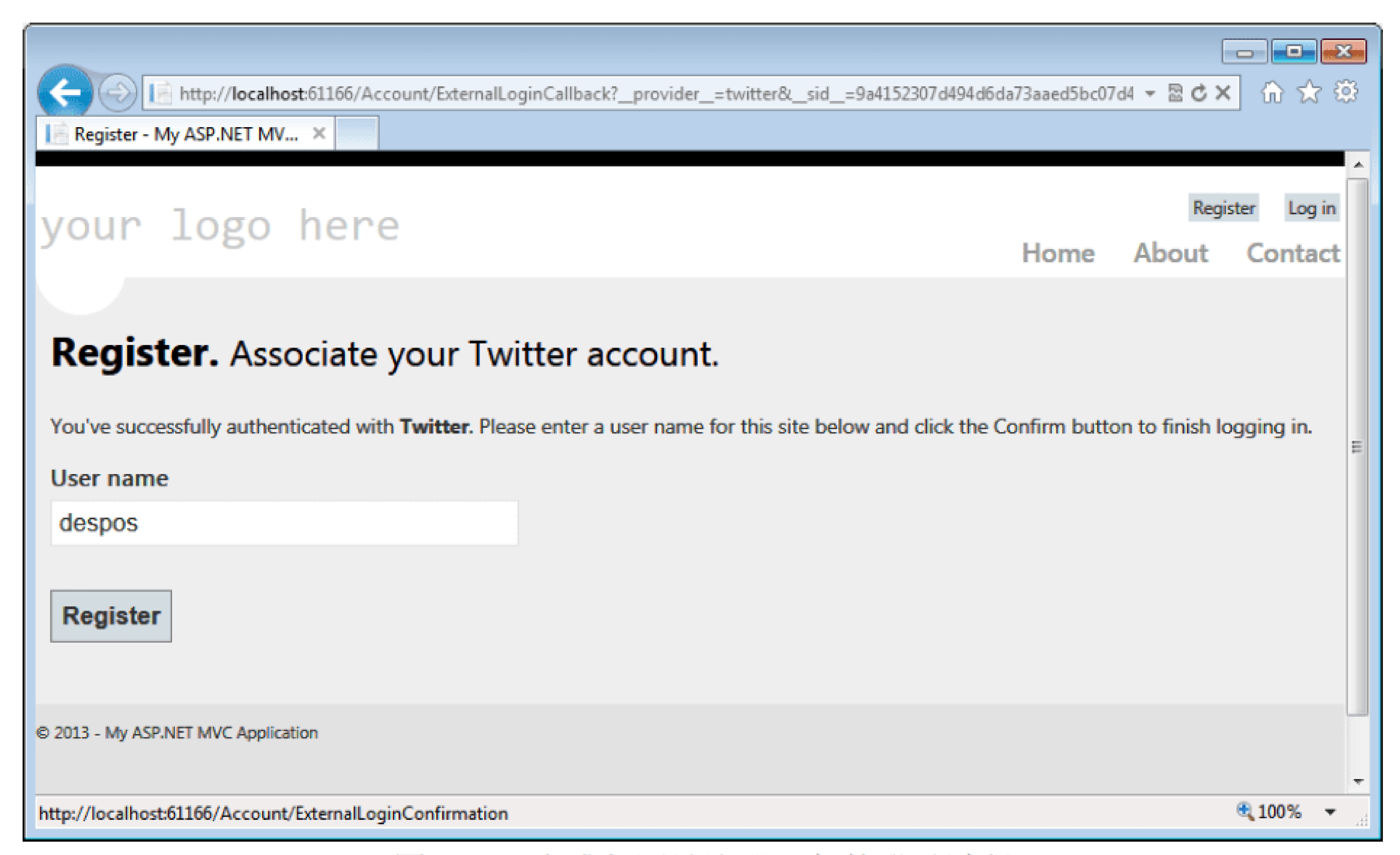


图 6-12　完成与网站注册一起的登录过程

在这一阶段输入的名称用于创建 ASP.NET 身份验证 cookie，它将明确地结束这一操作闭环：用 Twitter 核实凭据、要求用户输入其显示名称、并创建一个常规的身份验证 cookie。从现在起，ASP.NET 中需要身份验证的网站就可以开始正常工作了。

重要提示：

不论你是否决定合并登录和注册，你都有必要用 OAuth 提供程序(如果有的话)所返回的用户名称设法创建一个经典的 ASP.NET 身份验证 cookie。

该应用程序示例，在很大程度上是基于默认的 ASP.NET MVC 项目模板的，将用户数据保存到 App_Data 文件夹下的一个 MDF 本地数据库中。这个表通过使用简单的从 Web Pages 框架继承来的成员资格 API 来进行管理。

下面的代码显示了示例项目模板是如何检索如图 6-12 中所示的用户显示名称的。

```
var loginData = OAuthWebSecurity.SerializeProviderUserId(
```

```
        result.Provider, result.ProviderUserId);
    var name = OAuthWebSecurity
                .GetOAuthClientData(result.Provider)
                .DisplayName;
    return View("ExternalLoginConfirmation",
            new RegisterExternalLoginModel {
                UserName = result.UserName,
                ExternalLoginData = loginData
            });
```

对 GetOAuthClientData 的调用可以让你访问 Twitter 提供程序共享的有关登录用户的信息。接着，ExternalLogicConfirmation 视图会提供图 6-12 的实际标记。当用户单击注册时，会发生另一个对账户控制器的回调，具体来说是对 ExternalLoginConfirmation 方法进行回调。在此方法的主体中，有两件关键事情发生，归纳如下：

```
OAuthWebSecurity.CreateOrUpdateAccount(
    provider, providerUserId, model.UserName);
OAuthWebSecurity.Login(
    provider, providerUserId, createPersistentCookie: false);
```

第一行会为应用程序的成员资格本地数据库设置新的记录。第二行会实际创建身份验证 cookie。默认模板提供了大量的数据库表，比如 UserProfiles 和 webPages_OAuthMembership。webPages_OAuthMembership 表会用提供程序的名称(比如 Twitter)、提供程序用于用户的唯一 ID 以及指向内部 ID 的指针来存储记录，指向内部 ID 的指针能够以用户自己在图 6-12 中的页面上所选择的显示名称来唯一标识 UserProfiles 表中的用户。

重要提示：

要测试 Twitter 身份验证，需要一对真实的用户键值/用户密钥，但不一定非得从公共 Web 服务器上测试。可以从自己的 Visual Studio 环境从容地进行测试，并使用 localhost:port 作为回调 URL 的根目录。

6. 超出身份验证的内容

如前面各节所述，虽然大多数网站只使用社交网络来验证用户身份，但仍然有很多可以由网站自己去做的事情。在用户的凭据通过验证以后，几乎所有的外部提供商(当然也包括 Twitter 和 Facebook)都会返回一个称为访问令牌的含有字母和数字的字符串。

访问令牌是一个重要的信息，因为它会授予网站在用户要求和授权的范围内代表用户对社交网络进行操作的权限，正如图 6-9 中所描绘的。简而言之，如果只是为了对用户进行身份验证，则采集和存储访问令牌并不重要，但如果希望提供更多的功能，例如代表用户发帖或检索更多个人信息，那么采集和存储访问令牌就变得很重要了。

注意：

访问令牌不可能永远存续下去。例如，Facebook 令牌往往很快就到期，而 Twitter 令牌持续的时间稍长一些。如何处理访问令牌需要具体问题具体分析，但一般情况下，好的策略是在每次用户登录时保存访问令牌。可以将其保存到会话状态，或者更好的是，保存到一个数据库。

如何检索访问令牌呢？

访问令牌可以在 ExternalLoginCallback 方法中找到；即，身份验证成功后出现在你代码中的首个入口点。具体来说，可以在 AuthenticationResult 对象的 ExtraData 属性中找到它。

```
if (result.ExtraData.Keys.Contains("accesstoken"))
{
   // Save the access token for later use: result.ExtraData["accesstoken"]
   ...
}
```

例如，可以使用访问令牌检索用户的更多信息。请记住超越简单身份验证的功能都需要针对特定社交网络的专用软件开发工具包(SDK)。要与 Twitter 进行交互，你可能要用 TweetSharp。而对于 Facebook，最好的选择是用于 C#的 Facebook Client SDK。这两个库都很容易通过 NuGet 访问。下面是一个用于抓取 Facebook 用户额外信息的代码段：

```
var client = new FacebookClient(accesstoken);
dynamic user = client.Get("/me", new { fields = "first_name,last_name,email" });
```

使用类似的语法，可以访问当前用户的社交信息记录。更多的相关信息可以参考所选择的社交网络站点上的开发人员页面。

6.4　本章小结

安全性总是被看作 Web 应用程序的热门话题。因此几乎任何有关 Web 技术的课程或书籍都会单独划出一个章节来讲述如何编写安全应用程序。假如某个人了解基本的安全性以及用于经典 ASP.NET 的 Forms 身份验证(比如 Web Forms)，那么在 ASP.NET MVC 中也就没有太多要学习的了。

运行时管道与 Web Forms 中的相同，并且信任级别和进程标识也是以完全相同的方式建立的。此外，Forms 身份验证也是通过 HTTP 模块和一个高度可配置的 cookie 以相同的方式运行的。可以参考我的一本有关 ASP.NET 的书 *Programming ASP.NET 4*(微软出版社，2011 年出版)，其中的第 19 章有关于这些话题的深入讨论。

特定于 ASP.NET MVC 的是，在其中限制对操作方法和控制授权访问的方式。本章的前

半部分涵盖了这些内容。后半部分谈及，现在越来越多的应用程序集成了通过外部服务进行用户身份验证的队伍。可以通过一些单点登录方案和底层协议——OpenID 与 OAuth 来实现这一点。尽管 OpenID 基本上能用于唯一标识用户，但 OAuth 却可以做得更多。OAuth 可以从用户那里获得由服务提供商所持有的应用程序能够使用的资源的权限。这一章展示了一个使用流行的 Twitter 社交网络进行用户身份验证的 OAuth 应用程序示例，它可以为自己获得读取 Twitter 文以及连接到已登录用户的关注者的权限。

第 7 章

设计 ASP.NET MVC 控制器的注意事项

计算机非人性的一部分表现是，一旦完成编译并且顺利运行，它将忠实地完成工作。

——*Isaac Asimov*

控制器是在 ASP.NET MVC 中执行任何操作的核心元素。控制器负责获取提交的数据、执行相关的操作、然后准备和请求视图。这些看似简单的步骤通常会生成大量的代码。更糟的是相似的代码最终会被用到相似的方法中，相似的帮助器类却没有地方放置。

ASP.NET MVC 会保障更易于编写清洁和易于测试的代码的基础。可以肯定的是，ASP.NET MVC 基于的基础架构使这一保障有了实现的可能，并且比在 Web Forms 中更容易实现。但是仍然有很多工作留给了开发人员、及其编程准则和设计构想。

从架构上来说，控制器类似于 Web Forms 中的代码隐藏类。它是表示层的一部分，以某种方式存在以便将请求发送到应用程序的后端。如果缺乏了开发准则，控制器会很容易凌乱地增长并且无法理清，就像老式的代码隐藏类一样。所以，仅仅选择 ASP.NET MVC 并不能确定你能否保证代码的整洁度和质量。

在本章中，我们将探讨一个对 ASP.NET MVC 的设计方法，该设计方法会简化你需要机械化实现控制器类的步骤。其理念就是使控制器成为一个至精至简的类，以委托所肩负的任务而不是靠其自身去组织编排任务。这种设计对应用程序的其他层以及 ASP.NET MVC 基础架构的某些部分均有影响。

7.1　打造你的控制器

微软 Visual Studio 会让你很容易地创建自己的控制器类。为此，只需要右击当前 ASP.NET MVC 项目中的项目文件夹，并添加一个新的控制器类即可。在控制器类中，每一个落在控制器职责范围内的用户操作都会有一个方法。那么操作方法是如何进行编码的呢？

注意：

ASP.NET MVC 的 Visual Studio 工具让你不再仅仅局限于在 Controllers 文件夹中添加控制器类。可以在任何地方放置控制器类，并且右击任何一个文件夹都将显示创建新的空白控制器类的命令。

操作方法会收集输入数据，并将其用于准备对由应用程序中间层所公开的某些端点进行一次或多次的调用。接着，它会接收输出数据，确保输出数据的格式正是视图所需要接收的。最后，操作方法会调出视图引擎以呈现某个具体的模板。

可能所有这些加起来几行代码的工作也会使得即便是带有几个方法的控制器类变成一个混乱的类。首先——获取输入数据——主要由模型绑定器类为你处理。视图的调用只是调用一个能触发处理操作结果的方法。操作方法的核心位于执行任务和为视图准备数据的代码中。

7.1.1　选择正确的原型

一般来说，操作方法可能有两种角色：可以是一个控制器，也可以是一个协调器。“控制器”和“协调器”这样的字眼是从哪里来的呢？很明显，在这种上下文背景中“控制器”这个词与 ASP.NET MVC 控制器类并无关联。

这两个词指的是对象原型，一个来自于被称为职责驱动设计(Responsibility-Driven Design，RDD)方法的概念。通常情况下，RDD 适用于系统上下文中对象模型的设计，但它的一些概念同样适用于相对比较简单的操作方法行为建模的问题。

注意：

RDD 的更多相关信息，请查阅 Rebecca Wirfs-Brock 和 Alan McKean 合著的 *Object Design: Roles, Responsibilities, and Collaborations* 一书(Addison-Wesley，2002 年出版)。

1. RDD 概览

RDD 的本质在于将系统功能分解成系统应当执行的若干操作。接着，每一个操作会被映射到正在进行设计的系统的一个对象。执行该操作就变成了这个对象的具体职责。对象的作用取决于它所承担的职责。表 7-1 描述了 RDD 的主要概念，并且定义了一些与其使用相关的术语。

表 7-1　标准 RDD 概念和术语

概　　念	描　　述
应用程序	是指相互连接和相互作用对象的集合
协作	是指共同运行以提供一些有意义行为的两个对象之间建立的关系。协作这个术语是通过显式协议定义的

(续表)

概　　念	描　　述
对象	是指实现一个或多个角色的软件组件
角色	是指承担一个相关职责的软件组件的特性
职责	是指对象的预期行为。原型用于归类职责

表 7-2 总结了对象职责的主要分类。它们统称为对象角色原型。

表 7-2　标准 RDD 原型

原　　型	描　　述
控制器	组织安排其他对象的行为，并决定其他对象应该做什么
协调器	由事件请求，它会给其他对象委托任务
信息存储器	存储(或知晓如何获取)信息并提供信息
交互器	表示实现对象之间通信的外观模式
服务提供程序	针对请求执行特定操作
结构器	管理对象之间的关系

在 RDD 中，每个软件组件都会在特定的情形下发挥作用。当使用 RDD 时，你要运用原型为每一个对象指派其角色。我们看看如何将 RDD 原型应用到操作方法上。

2. 对请求的执行进行分解

我已经描述了所有操作方法要实现的一些常见步骤。我们可以把操作方法的职责分解如下：

- 获取随请求发送的输入数据
- 执行与请求相关的任务
- 准备用于响应的视图模型
- 调用下一个视图

可以使用控制器也可以使用协调器 RDD 原型来实现操作方法——但两者产生的效果不一样。

3. 充当“控制器”

让我们考虑一个简单的订购用例中的操作方法。在现实世界中，下订单绝对不只是像你在入门级教程中所看到的那样，往 Order 表中添加一个记录那么简单。

下订单的操作通常涉及几个步骤和对象。例如，它可能需要查询数据库，找出订购货物

的库存。还可能需要向供应商订购一个单独的订单来补充库存，并且通常需要检查客户的信用状况，以及与客户的银行和运输公司进行协同。最后，它还涉及对某些数据库表进行更新。当订购成功以后，系统会返回以便向用户显示订单 ID，也许要打印发票，还可能要估计交货日期。

下面的伪代码会让你对需要采取的具体步骤有一个概念：

```
 [HttpPost]
public ActionResult PlaceOrder(OrderInfo order)
{
    // Input data already mapped thanks to the model binder

    // Step 1-Check goods availability
    ...
    // Step 2-Check credit status of the customer
    ...
    // Step 3-Sync up with the shipping company
    ...
    // Step 4-Update databases
    ...
    // Step 5-Notify the customer
    ...

    // Prepare the view model
    var model = PlaceOrderViewModel { ... };
    ...

    // Invoke next view
    return View(model);
}
```

最起码，对所有这些步骤在控制器中进行编码意味着你最终要从表示层对数据访问层进行调用。对于简单的 Create、Read、Update、Delete(CRUD)应用程序来说是可以接受的，但对于更复杂的应用程序来讲就不可接受了。

即便每个列出的步骤可以用一行或两行代码解决，但你仍然会面临一个长得无法控制的方法。应用于 ASP.NET MVC 控制器类的 RDD 控制器原型会提示你使用以前的代码结构。但这即使对于不那么复杂的应用程序来说也未必是理想的选择。

重要提示：

用例实现背后的业务流程会涉及可能属于系统架构中不同层的多个组件。某些步骤会通过域名服务(比如准备发票)来实现；其他步骤可能要由外部服务(例如，从合作运输公司的关联系统获取交货信息)来实现；更多的步骤可能只是数据库的调用(获得订单 ID)。所有这些调

用都应被视为由表示层所触发的单个工作流步骤。

4. 充当“协调器”

RDD 协调器会建议你将所有这些步骤组织在单个工作线程对象中，形成操作的实现。从操作方法内，对工作线程进行单个调用并将其输出的数据用于视图模型对象。下面是其结构：

```
 [HttpPost]
public ActionResult PlaceOrder(OrderInfo order)
{
   // Input data already mapped thanks to the model binder

   // Perform the task invoking a worker service
   var service = new OrderService();
   var response = service.PerformSomeTask();

   // Prepare the view model
   var model = PlaceOrderViewModel(response);
   ...

   // Invoke next view
   return View(model);
}
```

现在 ASP.NET MVC 控制器方法的总体结构简单得多了。由传入的 HTTP 请求提出请求，操作方法将大部分任务转发给协调后面所有步骤的另一个组件。我把这些组件称为工作线程服务，或直接叫做应用程序服务。

重要提示：

工作线程服务或应用程序服务属于系统的应用程序层。应用程序层是实现从用例中所产生的应用程序逻辑的地方。这一层不能重用，因为它是特定于应用程序(和前端)的。可重用性要推至下一层的域层中。意即核心功能是可重用的(也就是域)，但呈现工作流会特定于应用程序。这里有一个简单的例子来说明这一点：在台式机前端，你可能有一个表单来获取所有数据，而在移动前端你可能需要经过多个表单，每个表单可能都需要与后端进行一些交互。核心功能(比如下订单或与运输公司协同)保持可重用性；工作流需要加以调整。

7.1.2　精简的控制器

ASP.NET MVC 是一个旨在方便测试和推动像关注点分离(SoC)和依赖性注入(DI)这样的重要原则的框架。ASP.NET MVC 会通知你应用程序被分隔为称作控制器的部分和称作视图的部分(更不用说这里讨论的模型了)。被强制创建一个控制器类并不意味着你就能自动取得

适当的 SoC 水平，也不意味着你要编写测试代码。正如第 1 章“ASP.NET MVC 控制器”中所述，ASP.NET MVC 给了你一个好的开始，但后面(所需)的分层取决于你。

我可能还没有说得太清楚的是，如果不够注意的话，则可能最终得到的是一个臃肿和凌乱的控制器类，它当然也不会比混乱(和让人看不上眼)的代码隐藏类更好。所以，你应该尽量将控制器类创建得像精简和平均的端点集合一样，并移除其中的任何累赘。

注意：

按照我的标准，早些时候我并不确定是否应该把 DI 看作一项原则。具体地说，DI 只是用于实现依赖反转原则的受欢迎模式，据此，从属类之间的接触面应该始终是一个接口而不是实现。比 DI 更少为人所知(和理解)的是，依赖反转原则是流行的 SOLID(单一职责、开闭原则、里氏替换、接口隔离以及依赖反转原则)首字母缩写中的“D”，SOLID 总结了编写整洁、高质量代码的五个关键设计原则。

1. 精简总是更好的

如果有一个长约 100 行逻辑的方法，该代码就可能包含 10 到 15 行的注释。一般而言，10%的代码注释被认为是一个合理的比例；如果希望为所做的解释清楚来龙去脉，切实帮到在你之后处理这些代码片段的人，我觉得注释比例甚至可以高达每三行逻辑一个。

然而，不论你如何决定理想的比例，我认为一个长达 100 行的方法其实没有太大意义。你也许可以将其分成 3 个或 4 个小一些的方法，并丢弃一些评论。

我不认为自己是软件度量方面的专家，但我常常会将我的方法保持在 30 行以内——这多少符合典型便携式电脑上 Visual Studio 编辑器中的显示空间。如何保持操作方法代码的短小精悍呢？让人惊讶的是，应用 RDD 协调器是必须要做的，但仅此还不够。

注意：

我分解代码和提高可读性所遵循的快速经验法则模式是，应该避免使用正常大小字体编写的每个方法需要在 Visual Studio 窗口中滚动才能显示完整的情况。

2. 作为视图模型构造器编码的操作方法

被设计成协调器的方法会在工作线程对象上调用一个方法、完成一些工作，然后取回一些数据。这些数据要直接装进一个字典或强类型的类，然后传给视图引擎。

然而，工作线程类会试图在中间层上的数据模型——即域模型——与表示层中的数据模型——即视图模型、或当前在视图中处理的数据之间做桥接(顺带说一句，“当前在视图中处理的数据”最初是在 MVC 文件中用来定义模型角色的措辞)。

如果在中间层调用的业务对象返回域对象的集合或聚集，则可能需要将此数据捏合到准确表示约定用户界面的视图模型对象中。如果把这项任务移到控制器类中，就又回到原点了。

通过使用工作线程服务和 RDD 协调器原型所截取掉的代码行，被大量用于构建视图模型的行替换了。

为了支持你获得精简的控制器，我推荐使用基于以下这几点的策略：

- 将所有操作传递给特定于控制器的工作线程服务类(部分应用层且不可重用)。
- 使工作线程服务类方法能接受来自模型绑定器的数据。
- 使工作线程服务类方法能返回表示为准备传给视图引擎的视图模型对象的数据。
- 通过特性捕获异常。
- 用.NET 代码约定检查先决条件，并在合适的情况下确保后置条件。
- 对于其他任何需要额外逻辑的情况，请考虑使用自定义的操作筛选器。

下面来看看我是如何设想实现一个工作线程服务类的。

3. 工作线程服务

工作线程服务是与控制器密切相关的一个帮助器类。可以合理地认为每个控制器都有一个不同的工作线程服务类。另一方面，工作线程服务只是控制器的一种扩展，并产生于 RDD 协调器角色所推动的控制器行为的逻辑分离。

我在这里使用服务一词是要表明这个类为调用方提供了一种服务；这与你可能了解的像 Windows 通信基础(Windows Communication Foundation，WCF)这样的实现服务的所有技术无关。同时，如果决定把应用程序层扩展到多台计算机，那么 WCF 就是将逻辑工作线程服务转变成实际 WCF 服务的绝佳技术。

图 7-1 显示了 ASP.NET MVC 中的工作线程服务的架构透视图。

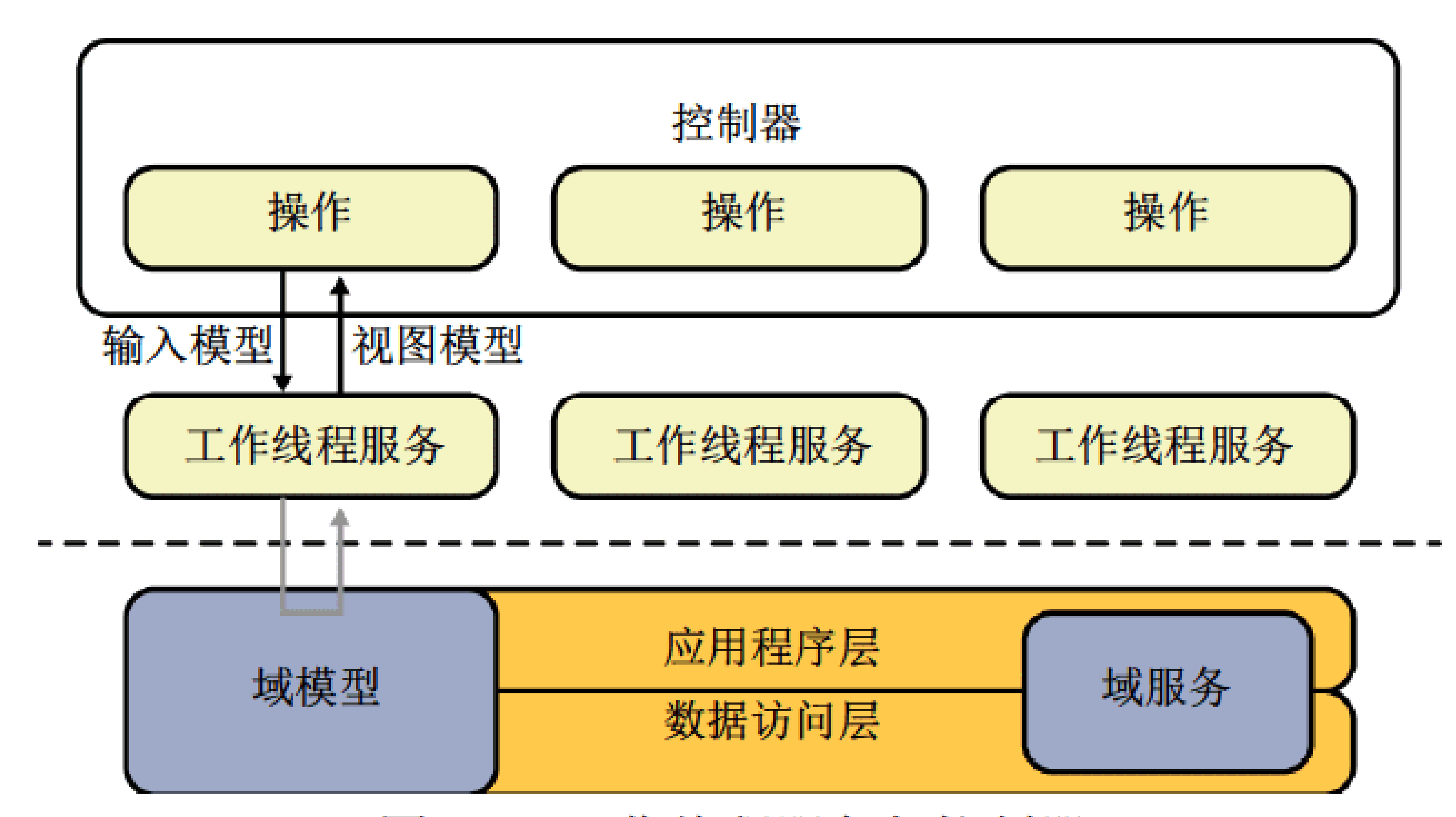


图 7-1　工作线程服务与控制器

工作线程服务只是关于设计的问题，而设计就是设计，无关其复杂性。所以，你不必非得等到一个大型项目才可以尝试使用这些功能。让我们通过一个简单的示例来展示工作线程服务方式的能力。诚然，对于一个简单的演示来说它可能显得有很多工作要做，但最终，你只需要一个额外的接口——它就能根据域的复杂程度良好地扩展。

4. 实现工作线程服务

首先可以在 ASP.NET MVC 项目中创建一个 Services 文件夹。建在哪个文件夹中取决于你自己。我通常会为每个控制器配备一个文件夹，外加一个额外的用于接口的文件夹。图 7-2 显示了一个采用这种方式创建的项目。

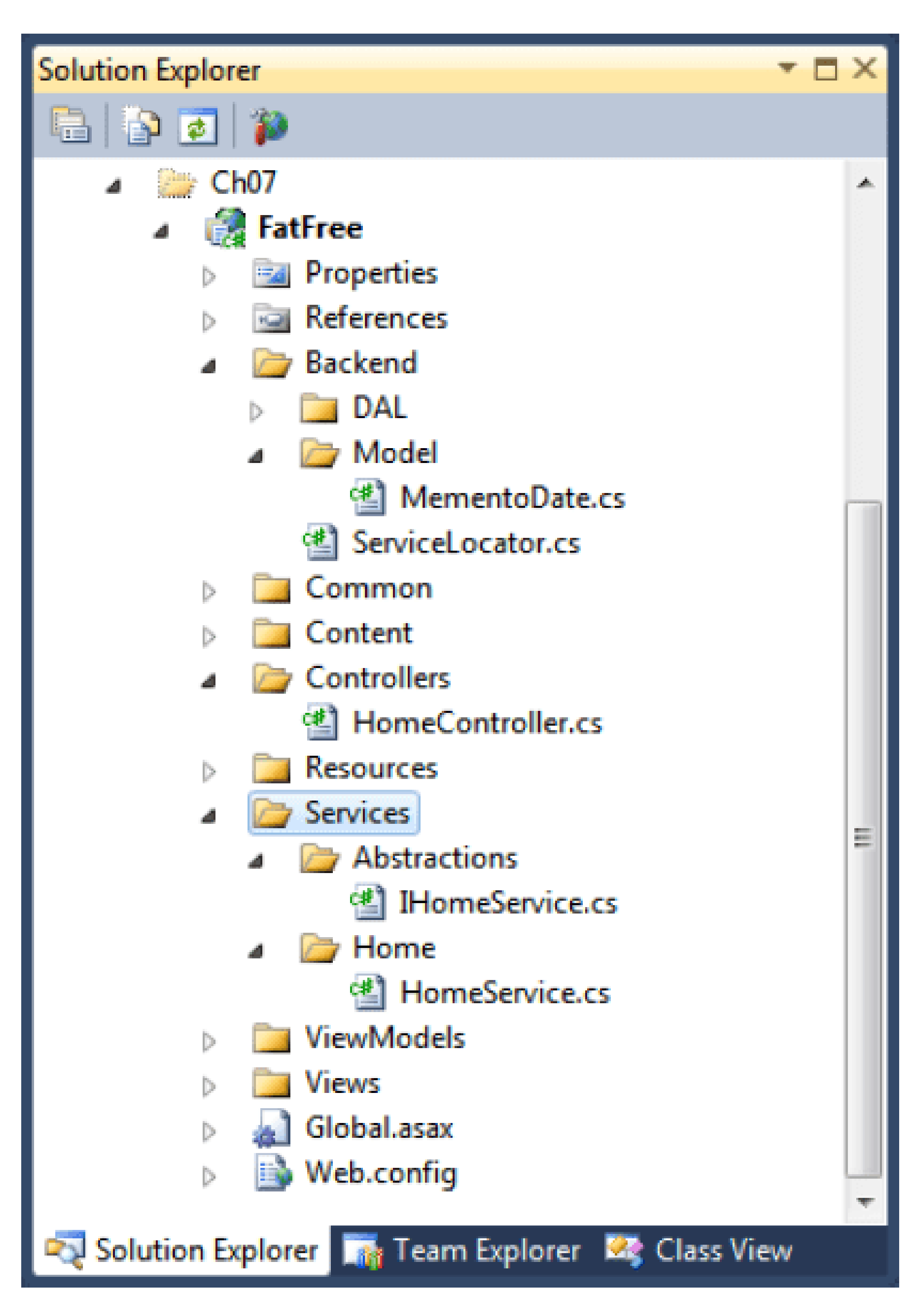


图 7-2　ASP.NET MVC 项目中的工作线程服务

如果愿意，可以将 Services 部分移到一个单独的程序集——这取决于你。如前所述，可以为每一个控制器创建一个工作线程服务。对于 Home 控制器，可以创建 IHomeService 接口和 HomeService 类，如下所示：

```
public interface IHomeService
{
    IndexViewModel GetIndexViewModel();
}
public class HomeService : IHomeService
{
    private IHomeService _homeService;
    public IndexViewModel GetHomeViewModel()
    {
       ...
    }
    ...
}
```

在示例应用程序中我们这样考虑，首页选出一个特别日期的列表，将这些日期与当前日期的时间跨度按天数呈现出来。在中间层，使用一个返回特别日期相关信息的存储库，比如日期、是绝对日期还是相对日期(例如，2 月 8 日，与年份无关)以及对日期的说明。下面是一个用于域模型的特别日期对象的示例：

```
namespace FatFree.Backend.Model
{
    public class MementoDate
    {
        public DateTime Date { get; set; }
        public String Description { get; set; }
        public Boolean IsRelative { get; set; }
    }
}
```

存储库在查询一些数据库时可能会形成这些对象的一个集合。总而言之，工作线程服务会获得一个 MementoDate 对象集合，并对其进行处理以获得一个 FeaturedDate 对象集合——一种属于另一个对象模型，即视图模型的类型。

```
namespace FatFree.ViewModels.Shared
{
    public class FeaturedDate
    {
        public DateTime Date { get; set; }
        public Int32 DaysToGo { get; set; }
        public String Description { get; set; }
    }
}
```

需要完成的有两个操作：首先，任何相对日期都必须转化为绝对日期，第二，指定日期与当前日期之间的时间跨度必须计算出来。例如，假设你要计算现在到下一个 2 月 8 日之间的跨度。如果正在计算 1 月 2 日或 3 月 5 日，则目标日期就是不同的。下面是工作线程服务中的一部分代码：

```
private IDateRepository _repository;
...
public HomeViewModel GetHomeViewModel()
{
    // Get featured dates from the middle tier
    var dates = _repository.GetFeaturedDates();

    // Adjust featured dates for the view
```

```
    //   For example, calculate distance from now to specified dates
    var featuredDates = new List<FeaturedDate>();
    foreach(var mementoDate in dates)
    {
        var fd = new FeaturedDate
                     {
                         Description = mementoDate.Description,
                         Date = mementoDate.IsRelative
                                ? DateTime.Now.Next(mementoDate.Date.Month,
                                                    mementoDate.Date.Day)
                                : mementoDate.Date
                     };
        fd.DaysToGo = (Int32)(DateTime.Now - fd.Date).TotalDays;
        featuredDates.Add(fd);
    }

    // Package data into the view model as the view engine expects
    var model = new HomeViewModel
                {
                   Title = "Memento (BETA)",
                   MessageFormat = "Today is <span class=
                   'dateEmphasis'>{0}</span>",
                    Today = DateTime.Now.ToString("dddd, dd MMMM yyyy"),
                    FeaturedDates = featuredDates
                };
    return model;
}
```

那控制器又如何呢？下面就是你需要的代码：

```
public ActionResult Index()
{
    var model = _homeService.GetHomeViewModel();
    return View(model);
}
```

图 7-3 显示了运行中的该示例页面。

可以看出，工作线程服务背后并没有神奇之处。顾名思义，它们就是分解代码的工作线程类而已，从逻辑上讲归属于请求的处理器——ASP.NET MVC 控制器。



图 7-3　工作线程服务处理日期

5. 我们真的需要控制器吗

没有哪个控制器方法的代码会像我这里展示的那样简单——只有一行逻辑。在真实的场景中，你可能需要将一些输入数据传递给工作线程服务；也许你会使用 if 语句快速排除一些情况，或者进一步编辑视图模型对象。后一种情况可能会在你试图获得一些可重用性和一个工作线程服务方法以满足两个或多个控制器操作方法的需求时发生。

要具体化操作方法中的代码，可以使用异常处理、null 检查和前置条件。最后，为了保持操作方法的至精至简，还要将 RDD 协调器的角色发挥到极限，把所有的处理逻辑移出控制器。

这是否意味着你不再需要控制器了呢？每个 HTTP 请求都会映射到一个操作方法，但你需要一些管道来实现这些连接。在 ASP.NET MVC 中，控制器只是基础架构的一部分，它不必包含很多代码，更加惊喜的是，也没有必要对其进行测试。如果把控制器看作是基础架构的一部分，自然会认可它们的基本作用；然而，你的工作线程服务却需要测试(这一点还会在第 9 章“ASP.NET MVC 中的测试与可测试性”中涉及，第 9 章将完全致力于介绍 ASP.NET MVC 应用程序的测试)。

6. 理想的操作方法代码

让我们通过分析一段应当在你的操作方法中找到的理想代码片段来为这次讨论画上圆满句号。它使用特性来处理异常和代码约定以确定先决条件。

```
[HandleError(...)]
public class DateController : Controller
{
    private readonly IDateService _workerService;
    public DateController() : this(new DateService())
    {
    }
    public DateController(IDateService service)
    {
        _workerService = service;
    }

    [MementoInvalidDateException]
    [MementoDateExistsException]
    [HttpPost]
    public ActionResult Add(DateTime date, String description)
    {
       Contract.Requires<ArgumentException>(date > DateTime.MinValue);
       Contract.Requires<ArgumentException>
          (!String.IsNullOrEmpty(description);

       var model = _workerService.AddNewDate(date, description);
       return View(model);
    }
}
```

在此示例中，自定义的异常特性用于捕捉可能由工作线程服务引发的特定异常。在这种情况下，你不必使用 if 语句和 null 检查来损坏你的代码(我并不反对使用 if 语句，但如果可以为自己和同事节省几行代码，并保持代码的可读性，那么我肯定会选择这样做)。

重要提示：

如果你是一个细心的读者，可能已经注意到了，我忽略了非常重要的一点：如何得到工作线程服务的实例。以及反过来，工作线程服务如何得到存储库中的实例？在代码中注入依赖性的技术和工具正是下一个话题(第 8 章，“自定义 ASP.NET MVC 控制器”涵盖了更多有关在整个 ASP.NET MVC 框架中注入点和相关技术的信息)。

7.2　连接表示层与后端

假设我们认同你不在控制器操作方法的上下文中编排用于给定请求的整个逻辑流程这一想法，那么要解决的下一个问题是你如何跨越应用程序表示层和后端之间的无形边界，以

及在哪儿跨越。图 7-1 中的细微虚线把工作线程服务块与应用程序层——应用程序后端的最顶层隔开了。

作为 RDD 协调器实现的操作方法，它会迫使你把请求转发到其他层，但在某些点上你需要越过边界去调用企业服务、数据库、业务组件、计算器以及你可能有的其他任何东西。因此，选择协调器路由对你如何组织下行层和应用程序层是有影响的。

注意：

层(layer)和层级(tier)这样的术语通常交替使用，有时候是有原因的。但是，一般来说，这两者是完全不同的实体。层是指逻辑上的分离，比如在相同的进程空间引入不同的程序集。而层级的是物理上的分离，例如，一个虽然可以获取到的软件模块，却驻留在不同的进程空间，并且也许是托管在不同的硬件/软件平台上的。要调用一个层级，需要序列化的数据、协议、并且还可能需要如.NET 空间中的 WCF 服务技术。

7.2.1　分层架构模式

人人都赞同多层系统在可维护性、易于实现性、可扩展性、可伸缩性和可测试性方面有着诸多优势。大部分时候，你要设置一个具有各种服务目标的三层架构，只为让每一层可能移动到不同的物理层。把逻辑层移动到自己的物理层上可以有许多原因：需要提高可伸缩性、需要更严格的安全措施，以及需要提高可靠性以防逻辑层因计算机故障而无法运行。

在三层方案中，通常有一个表示层部分，先负责处理用户输入然后安排响应；有一个业务逻辑部分，包括所有的函数算法和计算方法，使系统能够正常工作并与其他组件进行交互；还有一个数据访问部分，在其中可以找到从存储区读取和写入所需的所有逻辑。

虽然这种规划总体而言坚如磐石，但是根据现有的技术、研究成果和业界在模式和解决方案方面取得的进展来看，它也需要进行更新了。

1. 超越经典层

像表示、业务和数据访问这样的词在今天来看，好像包罗万象，又好像什么都不是，确实变得相当模糊。你如何真正地设计和实现它们呢？有太多的变量，太多的选择、模式和做法可以采用。接下来描述的分层架构模式就试图将这些部分扩展成更具体的东西，并为实现的过程提供指导。

在现代软件的架构中，你会发现表示层、应用程序、域和基础架构这些层。图 7-4 提供了一个分层架构的概览。映射到经典的表示+业务+数据层是相对简单的。经典的业务层常常会扩展为包罗各方面的内容：一些表示层的内容、一些基础架构的内容以及整个域。数据层位于基础架构中。这样做其实只是为了重申分层架构不是什么新东西；它只是以更现代和务实的方式应用那些虽然陈旧但仍可靠的理论。

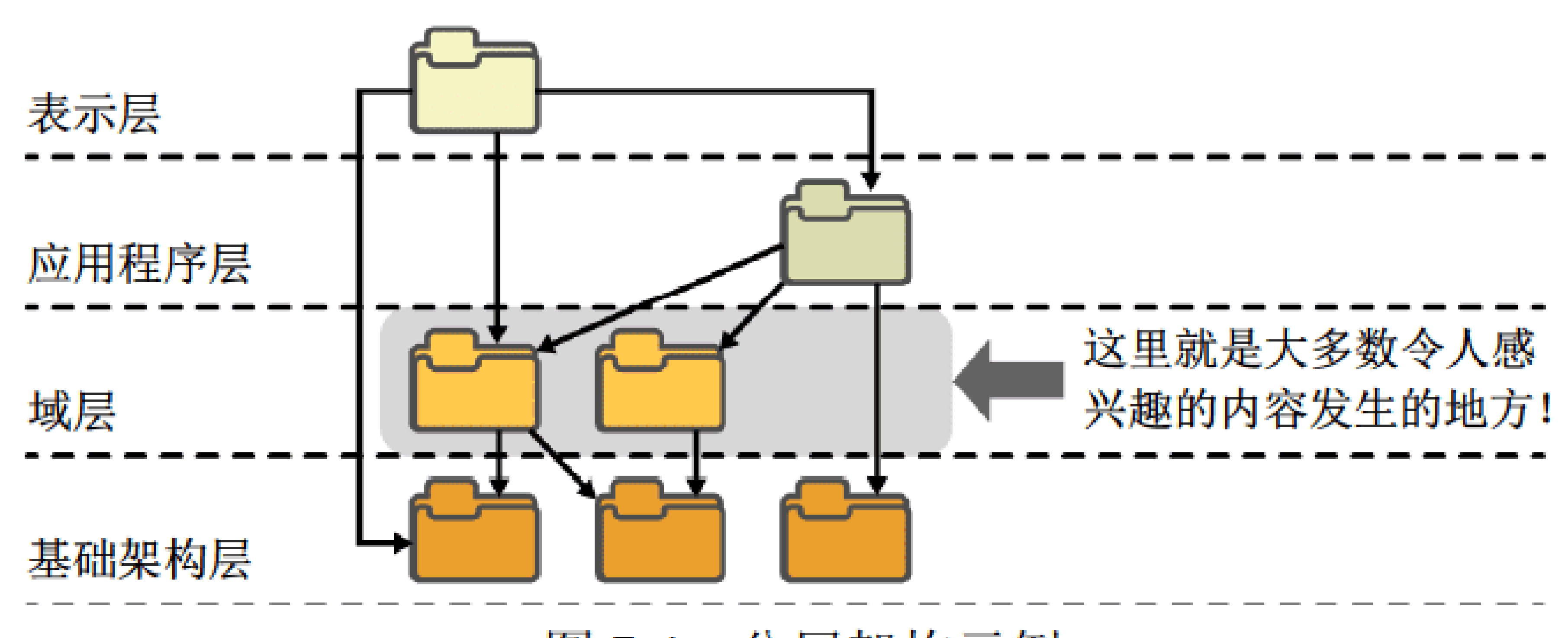


图 7-4　分层架构示例

我们可以总结出一个如下所示的分层架构：

表示层截获请求并将其传递到应用程序层。应用程序层(也称为服务层)是应用程序实现用例的区段。在这一点上，它特定于每个应用程序并且不可重用。应用程序层将端点公开给表示层，并将它从系统的其他部分解耦。应用程序层会安排域服务和外部服务，以及可能用到的特定业务的企业组件。最后，基础架构层会封装对数据的访问。

注意存储并不一定是关系型数据库。现在，它也可以是文档数据库(NoSQL)、云数据存储区或企业客户关系管理(CRM)系统。其多样性使得“数据访问”大抵已不再适合用来描述它了。

2. (惯用的)表示层

过去十年间软件架构师学到的重要经验是，不深入客户就不可能创建成功的应用程序。早在 2001 年，这就在“敏捷宣言”中作为一个关键点被提及，在现实中，客户与开发团队之间的协作往往在于当事人的良好意愿。但通常的情况是，协同合作只是一个大家并不会全力追求的良好愿望而已。

如今，软件用来指导用户尽可能以最简单有效的方式做他们的日常工作。软件必须转变成用户所期望的，而不是反过来——像过去多年的情况那样。我记得大约 25 年前与客户进行过一次讨论，我当时丝毫不难为情地说，“不，这个功能不可能按照你所建议的方式编程；我们使用的语言不支持这个。”这样的回答在今天难以想象。

无论你采用哪种技术构建应用程序客户端，表示层的代码部分都负责收集来自用户的输入数据和触发预期的行为。如果应用程序是分布式的，则表示层的代码段负责准备和执行远程调用以及在结果回来后安排新的用户界面。重要的一点是，表示层逻辑调用的是根据用户界面(UI)需要而设计的方法。这些方法接收和返回为 UI 格式化的数据。如今的规则是用户界面接收它所需要的，以它偏好的形式和形态。无论你碰到何种 UI 技术——微软 Windows、浏览器、HTML5、移动端、微软 Silverlight 等，这种方法都会使你成为大赢家。

面对实现的时候，表示层在这个意义上必然是大家所惯用的，即实际的代码取决于你正在使用的框架。虽然总体概念仍然一致，但表示层在 Web Forms 中是基于代码隐藏页面的，在 ASP.NET MVC 中是基于控制器(可选择性地加上工作线程服务)的，在 Silverlight 和 WPF

中是基于模型-视图-视图模型(MVVM)的，等等。例如，如果有一个完全的客户端单页面应用程序(SPA)，你会希望使用一些 JavaScript 框架，让你的代码能够使用 MVC 和 MVVM 模式。

就 ASP.NET MVC 而言，应用分层架构模式意味着将请求的响应结果委托给工作线程服务，接着，会联系后端获得响应。响应包含了表示层所需格式的数据。

(设备驱动)用户体验优先

从表示层开始你的设计已经日益变得重要。我喜欢把这种方式叫做用户体验优先(User Experience First，UXF)，也希望把这一概念与设备都应有自己量身定做的表示层的想法结合起来。关于这一点，我会在第 13 章“构建用于多种设备的站点”中阐述。

UXF 意味着你的系统设计始于用户实际交互的 UI、界面和工作流。由于软件是日常工作必不可少的工具，需要考虑的优化的主要形式——甚至在数据库之前——是在用户级别的工作流中消除瓶颈。

定义界面以后(过去，它们只是定义执行者和系统之间交互的 UML 用例图)，你就会知道要在体系架构中加载哪些数据以及输入哪些数据。这些都是现代架构的支点。

3. 应用程序层

应用程序层包含用例的实现，可以将其看作起编排作用的组件的聚合。这里的编排一词指的是服务于给定用例的算法实现。对于下订单用例来说，编排器是安排预期数据流和域服务(比如检查信用状况和回填库存)、外部服务(比如与运输公司进行协同)、业务组件(计算价格)，以及存储(更新内部数据库)的方法。

在相对简单的情况下，或如果正好没有具体的可扩展性要求，控制器的工作线程服务可能会与应用程序层相符。面对这种情况时，请牢记一个流行的面向服务架构(SOA)的警示：要健壮，不要繁杂(见图 7-5)。

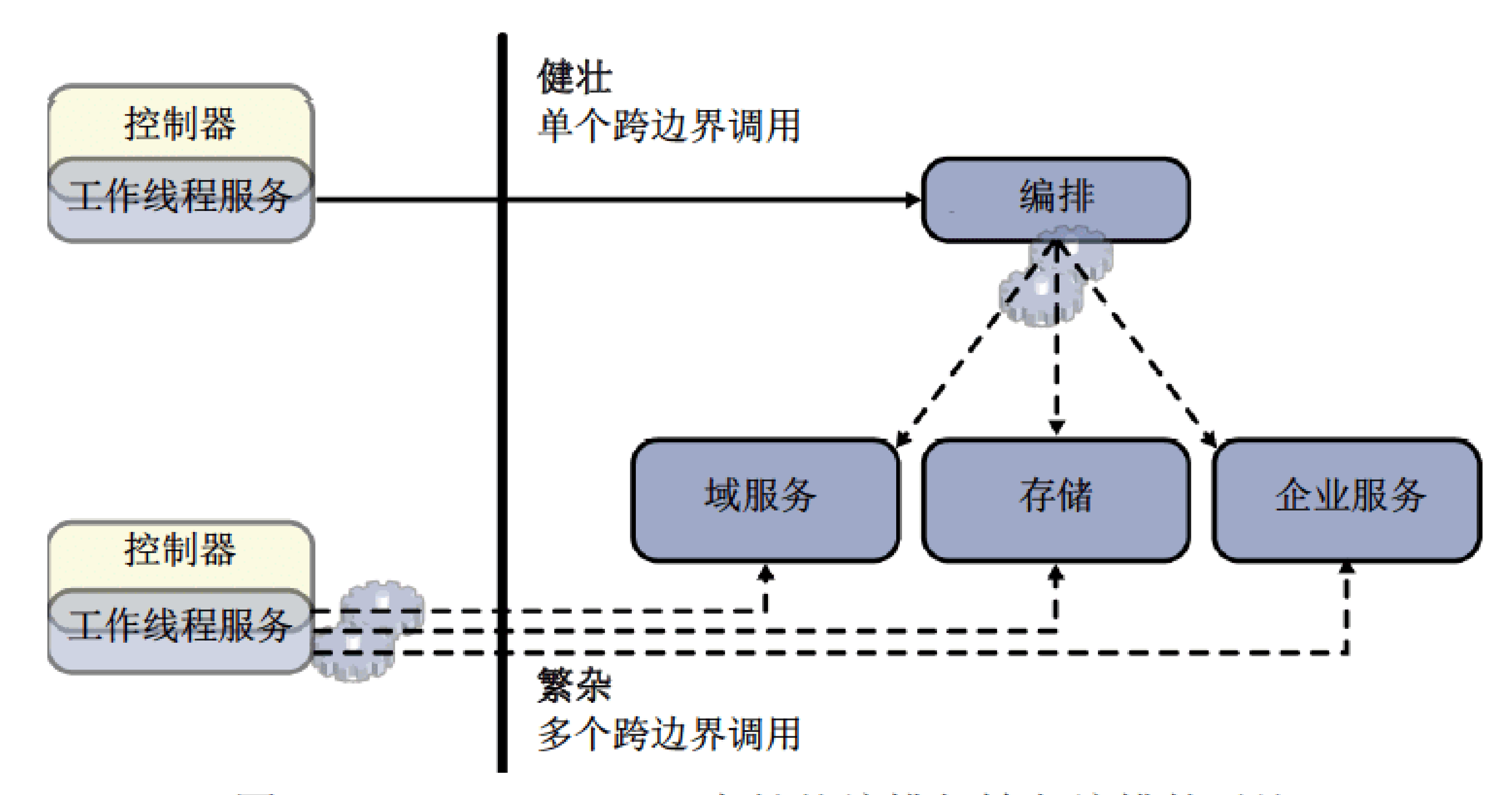


图 7-5　ASP.NET MVC 中健壮编排与繁杂编排的对比

如果编排的服务驻留(大多数情况下)在与工作线程服务相同的进程空间，你或许就不必引入另一个层。在这种情况下，编排是符合工作线程服务的。你可能希望将工作线程服务编译成一个单独的程序集，并把它们的逻辑等级从普通的帮助器类升级到架构构成块。

如果要编排的服务是远程的或者能够用于远程，你或许会希望在要编排的服务的空间中引入额外的编排层。在这种情况下，你要从工作线程服务直接以单个调用转到这个额外的层，该调用会将各个步骤所需的所有数据连接在一起。从另一个层——表示层——编排则会在分布式计算、序列化和网络延迟方面耗费你的成本。这就是反繁杂模式的 SOA 设计。

4. 域层

虽然还没有正式宣布淘汰，但 DataSet 对大部分开发人员来说已经是过去式了。并不是 DataSet 的设计有毛病，只是十年的时间在软件行业真的是太长了。技术的步伐就是这样的，10 年前令人信服的解决方案在下一个十年很难产生相同的效用。如今，实体框架——更不用提 NHibernate 和其他商业性解决方案——已经能快速创建一个基本的域模型，围绕它来设计你的应用程序了。可以利用实体框架和它的 Visual Studio 工具来创建实体和关系，并使它们呈现出在应用程序的眼中真实数据库的模样。你不需要成为域驱动设计(Domain-Driven Design，DDD)大师，也可以保持数据库详细信息不同于用来实现业务操作的对象。

现在很多开发人员发现创建实体模型变得简单有效，并可以用对象/关系映射器(O/RM)工具——比如实体框架或 NHibernate 将实体模型持久保存。

这将使你具有一个带有填充数据和行为的普通旧式 CLR(POCO)类的程序集。此对象模型代表了你在应用程序后端所拥有的数据模型。让具有描述性属性的类，比如 Customer，在逻辑上等效于某些数据库列：Name、Address、Website、Email、Contact、Orders。此外，Customer 类可能会有大量验证对象状态的方法，并实现特定于对象且与实例相关联的属性值上的操作。例如，你可能有一个方法计算与客户关联的订单(Orders)的总额。或者，对于具有 Date 和 PaymentMode 属性的 Invoice 类，可以使用一个返回预计付款日期的方法。在这个对象模型中，类对于持久性和连接字符串之类的事情一无所知。

注意：

对象模型、域模型和实体模型是常常可以互换使用的相似术语。但是，每个术语也都有自己独特的含义。有时候，没有到使用精确含义的地步时，你互换使用当然没错。然而，这种非正式的用法并不会抹杀每个词的真正含义。对象模型是普通、泛型的对象集合。域模型是特殊类型的对象模型，其中的每一个类都是 POCO、聚合根会被识别出、工厂会被用于构造函数、并且大多数时候值类型往往会取代基元类型。最后，实体模型基本上是一个表明类(大多是贫血类)的集合的实体框架术语，可能是 POCO，也可能不是。贫血类有数据，但几乎没有行为。

更加面向业务的行为，比如检查客户的信用状况或验证客户是否下了足够多的订单以获取高水平的服务或奖励。并且更主要的是，那些生产或操作跨多个实体且需要访问数据库的聚合数据的功能。这些都是限于域实体和其持久性的编排的特殊形式。需要你通过另一个称为域服务的类集合来实现它们。你可能有用于各个重要实体的域服务组件或者，用 DDD 术语来说，用于各个聚合根的域服务组件。域模型和域服务组成了域层。

5. 公开域的实体

域层中类的可见性是怎样的？它们应该出现在表示层中，还是注定要在后端系统层中存在与发展？如果人们可以在任何位置使用域对象的话，那么开发过程就会更加美好。如果你的特定情况能够通过使用一个对象模型将数据围绕域模型移动的话，那你就一定要庆幸自己幸运了，就这么做吧。然而，事实上，这除了会在会议演示和教程中出现，几乎从来不会出现在现实情况中。

表示层是围绕用例构建的，并且每个用例对实体的定义都可能会有所不同。订单在不同的用例中可能有不同的表示形式，如“用户发出了订单”和“用户检查待处理订单”。有时候，对 Order 实体使用相同的基于域的表示形式是可以的，因为不同的用例只需要一个原始 Order 的子集。但通常，每个用例需要的对象会选择一些来自原始 Order 的信息以及一些来自其他实体——例如 Products 的信息。在域模型中，你却正好没有这种聚合。因此，必须创建一个专注于视图的对象模型，并把数据转移到其中以及从其中转移出去。

专注于视图的对象模型是基于数据传输对象(data-transfer objects，DTO)的。DTO 是一个普通的容器类(只有数据，没有行为)，用于在层、层级之间甚或同一层之内传递数据。有了 DTO，你绝对可以在任何位置使用你所需的数据。但是，知易行难。基于 DTO 的解决方案十分昂贵，且编码阶段也比较痛苦。

要应对 DTO 的额外复杂性，可以利用像 AutoMapper(http://automapper.codeplex.com)这样的工具，使你免于编写重复的(和枯燥的)代码，或者可以使用 T4 模板来保存一些常见代码，自己只编写自定义的部分。

6. 基础架构层

如何得到一个对域对象的引用呢？一般情况下，域对象可以是瞬态的也可以是持久的。如果一个新的实例创建于内存中，并以运行时数据填充，我们就说它是瞬态的。如果实例包含从存储区读取的数据，我们就说它是持久的。当你要插入一个新的订单，通常处理的是瞬态实体；当你从存储区获取订单用于显示或处理时，你处理的就是持久实体。

基础架构层基本上就是数据库层，重命名后降低了放在数据和模型上的焦点和重点。在现代系统中，仍然需要持久性，但这不一定来自于关系模型，也仅限于原始数据的存储，所有一切(例如，约束、验证、操作)都在系统的更高层上发生。

因此，基础架构层处理持久性，它由存储库类构成，每个重要实体(或者，也可以称为聚合根)配有一个。存储库类使用一个指定的存储 API 来实现持久性。存储库类的实现在逻辑上属于数据访问层。存储库类会聚集多个服务于实体数据访问需求的方法。在存储库中，通常可以找到读取、添加、删除和更新数据的方法。

存储库向应用程序层公开接口并在内部使用存储 API。因此，可以有一个使用实体框架和 POCO 模型的存储库，或者一个使用 NHibernate 的存储库。也可以通过一个普通的 ADO.NET 层将存储库驻留在域中。还可以将存储库指向一些云存储、Dynamics CRM 或 NoSQL 服务。在每个实体的基础上(确切地说，只是主要实体)创建的存储库是通往实际持久层的大门。

使用存储库之所以重要还有一个原因——通过模仿持久层让你的业务服务具备可测试性。

存储库类的典型结构是什么？这就涉及两个主要流派。一部分人更愿意使用一个通用的存储库，为每个实体提供基本的 CRUD 方法，如下所示：

```
public abstract class Repository<T> where T : IAggregateRoot
{
   internal YourContext ActiveContext { get; set; }
   public Repository()
   {
      ActiveContext = new YourContext();
   }
   public void Add(T item)
   {
      this.AddObject(item);
      this.ActiveContext.SaveChanges();
   }
   public bool Remove(T item)
   {
       try {
          this.DeleteObject(item);
          this.ActiveContext.SaveChanges();
          return true;
       } catch {
          return false;
       }
   }
   public void Update(T item)
   {
       this.ActiveContext.SaveChanges();
   }
   :
}
```

就查询而言，下面是一个可能会在通用存储库类中看到的方法列表：

```
T[] GetAll<T>();
T[] GetAll<T>(Expression<Func<T, bool>> filter);
T GetSingle<T>(Expression<Func<T, bool>> filter);
```

传递该查询的详细信息以通过 GetAll 或 GetSingle 将该查询作为一个函数来执行。

另一部分人则倾向于直接使用一个常规的存储库类，其中包含要实现的逻辑所要求的多个方法。在这种情况下，最终会发现那是一个量身定做的实体类。可轻松地正确处理删除和插入操作的特殊情况，并使用一个特定的 get 方法来调用每个必要的查询。最终的选择决定于你，因为没有哪一个方法明显优于其他的方法。就个人而言，我喜欢有特定的(非泛型)存储库。

注意：

谈到存储库，还要提到一点。如果使用 O/RM 保留数据的话，它就关系到上下文对象的生命周期。在前面的代码清单中，你看见了一个 ActiveContext 属性被构造函数实例化。用这种方法，上下文的生命周期会与实例一样长。对相同实例的多次调用可以享有相同的身份映射，且可以跟踪更改。一个替代方法是使用会在每个存储库操作结束时丢弃的局部范围实例。

7.2.2　在层中注入数据和服务

使用层(和层级)的主要原因是 SoC。作为架构师，你要确定哪些层之间相互联系，并且要让测试、代码检查、也许还有签入策略来强制执行这些规则。然而，即使有两个层预期要协作，你也不会想让它们紧耦合。在这方面，依赖反转原则(回顾本章前面部分，这是 SOLID 首字母缩写中的“D”)能带来帮助。我甚至说过依赖反转原则比依赖注入模式更为重要，现在似乎人人都这样想了。

1. 依赖反转原则

如其定义，依赖反转原则(DIP)规定高级别的类不应依赖低级别的类。作为替代，高级别的类应该总是依赖于它们所需低级别类的抽象。在某种程度上，这一原则是面向对象设计支柱之一的一个特殊化：编程为一个接口，而非一个实现。

DIP 是以自上而下方式的形式来定义所有重要类方法的行为。在这种自上而下方式的使用过程中，你要专注在发生于方法级别的工作流上，而非其特定依赖性的实现上。然而在某种程度上，低级别的类应该连接到主流代码上。DIP 表明这要通过注入来实现。

在某种程度上，任何时候面对一个依赖性时，DIP 都会表示一个控制流的反转——只要主流程有权访问依赖性的抽象，它就不会去关注依赖性的细节。然后依赖性会在必要的时候以某种方式注入。图 7-6 显示了一个用于 DIP 典型示例的经典图的个性化版本，最早由

Robert Martin 在他的论文中提出，可以在 http://www.objectmentor.com/resources/articles/dip.pdf 处找到。

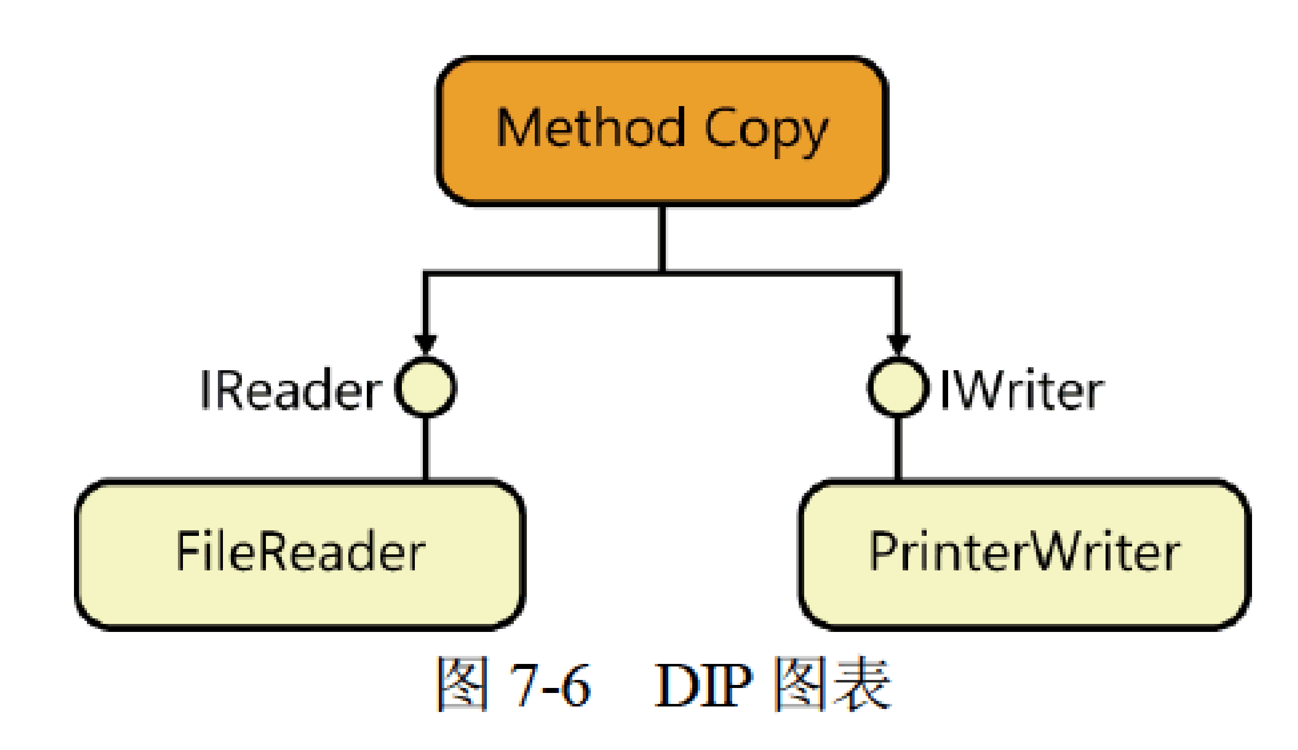


图 7-6　DIP 图表

该论文介绍了一个示例 Copy 函数，从一个源读取并写入到目标流。该 Copy 函数理论上不关注读取器和写入器组件的详细信息。它应该只会关心读取器和写入器的接口。之后读取器和写入器被注入，或围绕着 Copy 函数的实现以某种方式解析。这一点如何处理取决于你打算使用的实际模式。

要处理 DIP，通常可以使用两种模式：服务定位器模式和依赖注入模式。

2. 服务定位器模式

服务定位器模式定义了一个组件，它知道如何检索应用程序可能需要的服务。调用方无须指定具体的类型；调用方通常表示接口、基类型或者是以字符串或数字编码形式存在的服务昵称。

服务定位器模式会隐藏组件查找的复杂性，处理缓存或实例池，一般情况下，会提供一个组件查找和创建的常见外观。下面是一个服务定位器的典型实现：

```
public class ServiceLocator
{
   private static const String SERVICE_QUOTEPROVIDER = "quoteprovider";

   // You might also want to have a generic method GetService<T>()...
   public static Object GetService(Type t)
   {
     if (t == typeof(IQuoteProvider))
     {
        return new SomeQuoteProvider();
     }
     ...
   }

   public static Object GetService(String serviceName)
   {
```

```
            switch(serviceName)
            {
                case SERVICE_QUOTEPROVIDER:
                    return new SomeQuoteProvider();
                ...
            }
        }
    }
```

正如你所看到的，定位器仅仅是对知道如何获得指定(或间接引用的)类型实例的 Factory 对象的一个封装。我们现在来看看调用定位器的代码。下面的代码揭示了一个类，这个类首先为指定的符号列表获取引用，然后向一个 HTML 字符串呈现其值：

```
public class FinanceInfoService
{
  public String GetQuotesAsHtml(String symbols)
  {
    // Get dependencies
    var renderer = ServiceLocator.GetService("quoterenderer");
    var provider = ServiceLocator.GetService("quoteprovider");

    // Use dependencies
    var stocks = provider.FindQuoteInfo(symbols);
    var html = renderer.RenderQuoteInfo(stocks);

    return html;
  }
}
```

定位器代码存在于管理抽象的方法内，工厂是协议的一部分。只查看 FinanceInfoService 类的签名，你并不能获知它是否在外部组件上有依赖性。必须检查 GetQuotesAsHtml 方法的代码将其找出来。

服务定位器的重点是实现组件之间的最低可能耦合度。定位器代表了应用程序用来获得其所需所有外部依赖性的集中控制台。这样的话，服务定位器模式也就会产生令人愉快的意外结果，增加你代码的灵活性和可扩展性。

使用服务定位器模式从纯粹功能的角度来说不是一件坏事。然而，在实际中存在着更好的选择：依赖注入模式。

3. 依赖注入模式

服务定位器和依赖注入的最大区别是，使用依赖注入，工厂代码存在于正在运行的类之外。该模式表明你要以这样一种方式设计类：它从外部接收其所有依赖性。下面是如何重写

FinanceInfoService 类以使用依赖注入模式的代码：

```
public class FinanceInfoService
{
  private IQuoteProvider _provider;
  private IRenderer _renderer;

  public FinanceInfoService(IQuoteProvider provider, IRenderer renderer)
  {
      _provider = provider;
      _renderer = renderer;
  }

  public string GetQuotesAsHtml(string symbols)
  {
    var stocks = _provider.FindQuoteInfo(symbols);
    string html = _renderer.RenderQuoteInfo(stocks);
    return html;
  }
}
```

在类中使用依赖注入的时候，开发人员必须作出的一个关键决定是，如何以及在何处让代码注入。有三种方式可以把依赖性注入到类中：使用构造函数、一个可设置的属性、或一个方法的参数。这三种方式都有效，最终的选择取决于你。一般的共识是将构造函数用于必要的依赖性，setter 方法用于可选的依赖性。然而，仍有一些注意事项。

如果有很多依赖性呢？在这样的情况下，你的构造函数看起来会非常混乱。虽然构造函数中的一长串参数通常是某些设计问题的标志，但这也不是一个硬性规定。你可能会遇到复杂构造函数有很多参数的情况。这时，将依赖性分组到复合对象是一个解决方案。简而言之，你的目标应该是在构造时揭示依赖性和意图。可以有两种方式达成这一点：通过一套经典的构造函数，你要设法保持其尽可能的简单，或者通过工厂。

工厂是 DDD 方法的首选方式。使用工厂，可以将所需类型实例的上下文表达得更清楚。也可以在工厂代码内处理依赖性，并确保从一开始即返回有效的对象。此外，你的类最终只会有默认的构造函数(可能作为受保护的成员实现)。

使用构造函数也会妨碍继承关系，因为派生的类可能也需要接收依赖性。当你添加一个新的依赖性时，设计方案可能需要更多的重构工作。

然而，当依赖性是可选项的时候，并不会严格要求在构造函数层面显示该依赖性。这时，使用 setter 属性就好，而且很有可能这就是推荐的方法，因为它有助于保持构造函数(或工厂代码)的简洁。

总之，使用构造函数和使用 setter 属性都有各自的理由。与许多其他架构问题一样，正确的答案是，“看情况”。它也取决于你的个人喜好。

4. 用于控制反转的工具

DI 会将所有涉及依赖性设置的代码从类中除去。当依赖性嵌套时，代码可能会有许多行的长度；而且，大部分都是样板代码。为此，开发人员创建了称为控制反转(IoC)的特设框架。IoC 容器是一个专门创建用来支持 DI 的框架。可以认为它是能迅速有效实现 DI 的生产力工具。从应用程序的角度看，容器是一个富工厂，提供对要检索的且以后要使用的外部对象的访问权。

所有的 IoC 框架都是围绕一个容器对象构建的，当该容器对象绑定到某些配置信息时，它就会解析依赖性。调用方代码会将该容器实例化，并将所需的接口作为参数传递。在响应中，IoC 框架会返回一个实现该接口的具体对象。IoC 容器持有一个类型映射字典，通常会将一个抽象类型(比如接口)映射到一个具体的类型或指定具体类型的一个实例。表 7-3 列出了一些目前可用的最流行的 IoC 框架。

表 7-3　流行的 IoC 框架

框　　架	URL
Autofac	http://code.google.com/p/autofac
Castle Windsor	http://www.castleproject.org/container/index.html
Ninject	http://www.ninject.org
Spring.NET	http://www.springframework.net
StructureMap	http://structuremap.net/structuremap/index.html
Unity	http://unity.codeplex.com

对其配置以后，IoC 容器使你可以用一个调用就解决你类型之间整个依赖关系链的问题。此外，为你省去了内在依赖性的各种复杂问题。例如，如果在类的构造函数或属性中有一些 ISomeService 参数，就肯定可以在运行时得到它，只要你指示 IoC 容器来解析它。此方法的优点是，如果映射到 ISomeService 的具体类型构造函数有其自身的依赖性，这些也都会自动解析。

你会更进一步发现：使用 IoC 容器，便不用再担心依赖性带来的问题。另外，你只需要使用 IoC 所支持的语法来设计依赖性图表就可以了。其他一切都不用再操心了。

IoC 容器在其所支持的语法(比如 lambda 表达式)、使用的配置策略(比如外部 XML 方案)，以及可用的附加功能方面会有所不同。有两个特征如今的重要性很高，即：面向方面的能力(具体来说就是拦截)和促进与 WCF 服务集成的专用模块。

注意：

关于 Unity(微软的 IoC 库)，你发现其中包含有高级功能，包括我在 MSDN 杂志的前沿专栏中提到过的拦截功能；具体来说是在 2011 年 1 月/2 月的那一期。可以在 http://blogs.microsoft.co.il/blogs/gadib/archive/2010/11/30/wcf-and-unity-2-0.aspx 处找到一篇精彩的文章，它可以帮助你了解如何使用 Unity 在 WCF 服务的初始化过程中注入依赖性。

5. 穷人的 DI

如今，人们往往把依赖注入模式与使用 IoC 框架混为一谈。使用 IoC 框架是关于生产率的问题；因而，它通常需要最低临界值的复杂度以变得真正有效。在简单的情况下，可以选择所谓的穷人 DI。可以回看图 7-3，在 FatFree 示例的源代码中找到有关这种技术的例子。

你将如何在控制器类注入一个工作线程服务，又如何在工作线程服务中注入存储库呢？最明显的方法是让控制器和工作线程服务创建一个新的依赖对象的实例。但是，此路由创建的是一个对象之间的紧密依赖，却阻碍了可扩展性和可测试性。这里有一个更好的办法：

```
public class HomeController : Controller
{
    private readonly IHomeService _workerService;
    public HomeController() : this(new HomeService())
    {
    }
    public HomeController(IHomeService service)
    {
        _workerService = service;
    }
    ...
}
```

当使用默认的构造函数实例化一个控制器时，工作线程服务成员会指向一个新创建的默认工作线程服务类的实例。第二个构造函数可用于手动注入你想要注入的实例，至少是出于可测试性的原因。同样，在工作线程服务中注入存储库依赖也是同样的过程。

然而，ASP.NET MVC 通常会将默认的构造函数用于每个控制器类，除非你获得了对控制器工厂的控制权。

注意：

ASP.NET MVC 设计有几个扩展点，但它通常缺乏对 DI 的全面支持。服务定位器通过添加新的扩展点，可能是使现有框架能更松散地耦合在一起的最有效的方式，因为它是侵入性最小的解决方案。服务定位器充当了黑盒的角色，你把它安装在一个特定点，并让它知晓需要什么协议以及如何得到它。ASP.NET MVC 有自己的服务定位器模型，称为依赖解析器，

我将在第 8 章中讨论它们。

7.2.3　获得对控制器工厂的控制权

在 ASP.NET MVC 中，控制器类的实例化是一个受关注的时刻。ASP.NET MVC 基础架构包括一个工厂，该工厂使用所选控制器类的默认构造函数。如果控制器类上有参数化的构造函数，并且需要传入一些数据呢？这种情况没有现成的支持可用，但 ASP.NET MVC 良好的可扩展性设计为你提供了一个挂钩，可以用你自己的工厂取代默认的控制器工厂。

替换默认控制器工厂的一种常见方法是在其中集成一个 IoC 容器，这样就可以通过查看注册类型表来完美地解析任何参数。接下来的几节内容会介绍如何去实现的问题。

1. 注册一个自定义控制器工厂

首先要在 Application_Start 中注册你自己的控制器工厂。控制器工厂是实现 IControllerFactory 接口的类。要注册这个工厂，你要按照 ASP.NET MVC 推行的新模式创建一个适当的配置类(当然其他的方法也不错)。

```
protected void Application_Start()
{
    // Register a custom controller factory
    ControllerConfig.RegisterFactory(ControllerBuilder.Current);
    ...
}
```

ControllerConfig 类是一个带有静态方法的简单类，如下所示:

```
public static void RegisterFactory(ControllerBuilder builder)
{
    var factory = new UnityControllerFactory();
    builder.SetControllerFactory(factory);
}
```

让我们从内部了解一下控制器工厂。

2. 构建一个自定义控制器工厂

典型的控制器工厂类继承自 DefaultControllerFactory，并重写了一些方法，详情如表 7-4 所示。

表 7-4　流行的 IoC 框架

可重写的方法	描　　述
CreateController	管理控制器实例的创建。它需要控制器的名称(例如，home)并且返回控制器实例。默认实现首先会调用 GetControllerType 将控制器名称映射到类型并随后调用 GetControllerInstance 实际创建实例
GetControllerInstance	获取控制器类型以及请求上下文，并返回新创建的控制器实例
GetControllerSessionBehavior	获取控制器类型的会话状态行为。 注意：如果想要编程控制会话状态行为确定的方式？可以重写这个方法。默认情况下，它是由 SessionState 特性控制的
GetControllerType	获取控制器名称及请求上下文并返回控制器的预期类型。 注意：如果不喜欢控制器类型总是由“Controller”之前的名称指定这一惯例的话，可以重写这个方法
ReleaseController	当放弃控制器实例时，执行清除任务

注意，CreateController 和 ReleaseController 方法是公共的；其他的所有方法都是受保护的。创建自己的工厂也使你可以在新创建的控制器实例上执行任何类型的自定义工作。例如，你可能想要集中初始化一些自定义属性(如果从一个基类派生控制器的话)。更有可能的是，你可能想要使用这个挂钩为控制器实例提供一个手工的操作调用程序(我会在第 8 章中讨论操作调用程序)。

3. 基于 Unity 的控制器工厂

当引入一个基于 Unity 的自定义工厂时，至少你会想要重写 GetControllerInstance 以使用 Unity 基础架构来解析控制器类型，如下所示:

```
public class UnityControllerFactory : DefaultControllerFactory
{
   public static IUnityContainer Container { get; private set; }

   public UnityControllerFactory()
   {
      Container = new UnityContainer();
      Container.LoadConfiguration();
   }

   protected override IController GetControllerInstance(
                    RequestContext requestContext, Type controllerType)
   {
```

```
            if (controllerType == null)
                return null;
            return Container.Resolve(controllerType) as IController;
        }
    }
```

使用 Unity 引擎保证了植根于控制器类型的进一步依赖性得以识别和解析。参照 FatFree 的例子，这正是工作线程服务和日期存储库依赖性的情况。

刚才所示的代码看起来是非常通用，这也是 IoC 工具最终变得如此成功的原因。IoC 工具提高了生产力，因为它们为你节省了大量的样板代码。但是详细信息在哪里呢？

就像其他任何 IoC 框架一样，Unity 需要一些配置数据。可以在 web.config 文件中配置这些数据，也可以使用 Unity API 以编程方式添加。归根结底，这些配置数据包括了指定哪个接口类型映射到哪个具体类型，以及在哪里找到这些类型。下面是一个包含 Unity 节的修改过的 web.config 文件：

```
<configuration>
  <configSections>
    <section name="unity" type="Microsoft.Practices.Unity.Configuration.
     UnityConfigurationSection,
                              Microsoft.Practices.Unity.Configuration, ..." />
  </configSections>

  ...
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <assembly name="FatFreeIoC" />
    <namespace name="FatFreeIoC.Backend.DAL" />
    <namespace name="FatFreeIoC.Services.Abstractions" />
    <namespace name="FatFreeIoC.Services.Home" />
    <container>
      <register type="IHomeService" mapTo="HomeService">
      </register>
      <register type="IDateRepository" mapTo="DateRepository">
      </register>
    </container>
  </unity>
</configuration>
```

名称空间条目会注册一个 Unity 用来限定类型的名称空间。程序集条目列出了被认为会解析类型的程序集。在容器组内，可以注册类型。注册节点可以包含很多子节点，为注入和拦截提供额外信息(这些内容不属于本书范畴。要获知相关信息，请在 http://msdn.microsoft.com/en-us/library/ff663144.aspx 上参阅在线的官方 Unity 文档)。

7.3　本章小结

在这一章中，我试着用实体自己的名称和角色来调用每一个涉及典型 ASP.NET MVC 应用程序的实体。如果乍一看忍不住要跳过这一章，被似乎没必要的复杂性搞得心烦，我一点儿也不会觉得惊讶。究竟为什么应该创建一个工作线程服务呢，它显然是在浪费 CPU 周期啊?

如果你的应用程序比很多已经提供的指南应用程序(比如 Music Store)复杂，我认为是有好处的。你应该继续从操作方法调用实体框架数据上下文，并且不用管层和分层架构了。可以愉悦舒心地以这种方式设计很多应用程序。不应有忽视入门级指南的想法。它们做了大量的工作让你开始展示具体的内容。

但是，应该清楚那只是第一步。许多常见做法在模型和域的复杂性方面的伸缩性并不好。在这一章中，我总结了一些把你的 ASP.NET MVC 应用程序上升到下一个复杂性水平的实践。一个主要的好做法是通过将大部分逻辑移除出工作线程服务，让控制器类保持至精至简。另一个好的做法是通过把代码移到 HTML 帮助器和控制器，让视图维持在尽可能低的级别和被动性，最好是通过强类型的视图模型，但呈现操作也是可以接受的。另一个做法是将应用程序的后端分层，以区分域模型、域服务和存储库。存储库应该是访问数据的唯一方式，并且服务应该尽可能地调用它们，由此将控制器与它们分隔开。

那么，简易指南是演示这些内容的错误方式吗？不，只要你认为它们中的代码是本章(并且在本书的其余部分中也会作适度介绍)中讨论的更一般模式的一个特例。

对于简单的应用程序来说，ASP.NET MVC 是一个易用的框架；对于复杂的应用程序，它可能就有些棘手了。但是，如果掌握得好，生成的代码肯定会是高质量的——比 Web Forms 中的好得多，在技能和付出相同的情况下。

第 8 章

自定义 ASP.NET MVC 控制器

我们需要那些追求梦想的人，他们梦想的事物从未曾被梦想过。

——*John F. Kennedy*

整个 ASP.NET MVC 堆栈充满了扩展点。扩展点是代码中发生的实际行为可以从外部以及可替换的提供程序读取的地方。在前面几章中，你看到了一些扩展点的例子。例如，你看到了如何更换控制器工厂——即为每个传入请求返回新的控制器类实例的组件(参见第 7 章“设计 ASP.NET MVC 控制器的注意事项”)。第 4 章“输入表单”中讨论了如何替换模型元数据的提供程序——即读取要在 HTML 表单中使用的有关类的元信息的组件。

在像 ASP.NET MVC 这样的编程框架中，可扩展设计的好处会惠及到在其之上构建的所有应用程序。在这一章中，我的目标是帮助你发现可在 ASP.NET MVC 中找到的扩展点，并用几个例子进行说明。首先我会浅谈一下 ASP.NET MVC 中的扩展模型，然后在控制器方面进行详尽考察，这样你在闲暇时可以进行自定义且产生至精至简的控制器以及提升可维护性和可读性。

8.1 ASP.NET MVC 的扩展模型

扩展点与扩展模型是紧密耦合的。这是开发人员可以去除组件的现有实现并创建其自有实现的编程模型。在 ASP.NET MVC 中，你有两个在功能上等效的扩展模型，它们要求你使用不同的 API。其中一个基于你使用连贯语法显式注册的提供程序。另一个基于经典的服务定位器模式，它的引入让开发者从控制反转(IoC)框架中受益无穷。

这两个模型都完全支持了内部组件的替换，如视图引擎、操作调用程序、元数据提供者和工厂。ASP.NET MVC 扩展模型的另一部分基于特殊的特性——称为操作筛选器——用于在控制器方法的执行流中插入自定义(甚至是可选的)代码。我们首先来处理内部组件的替换。

8.1.1　基于提供程序的模型

ASP.NET MVC 堆栈的可替换内部组件是用于非常特殊的情形的。你不会希望在你编写的任何应用程序中利用它们。假设这些扩展点只在它们被需要的时候存在。如果在某个时候你决定替换它们，请注意你是在与 ASP.NET MVC 的内部机制进行交互。因此，如果出现性能或功能问题，你的组件会是第一个被质疑的对象。

1. 扩展点构成的库

表 8-1 提供了扩展点的简明列表。

表 8-1　可替换组件

提 供 程 序	接口或基类	描　　述
操作调用程序	DefaultActionInvoker	管理操作方法的执行以及浏览器响应的生成。它会与筛选器和视图引擎交互(稍后会介绍关于它的更多信息)
控制器工厂	DefaultControllerFactory	作为控制器实例的工厂提供服务。它可以用于为新创建的控制器实例设置自定义操作调用程序
字典值	IValueProvider	为请求读取要被添加到输入值字典的值。内置值提供程序会从查询字符串、表单、输入文件以及路由中读取。自定义提供程序可能会从 Cookies 或 HTTP 标头读取
模型绑定器	IModelBinder	定义将一些值提供程序字典中的原始值转换成特殊复杂类型的绑定器类型
模型绑定器提供程序	IModelBinderProvider	定义模型绑定器工厂，它会动态选择用于指定类型的正确模型绑定器
模型元数据	ModelMetadataProvider	检索与类成员有关的元信息，并将该信息与类型实例关联起来。默认提供程序会从 DataAnnotations 特性中读取元信息
模型验证器	ModelValidatorProvider	这一提供程序的描述与上面的模型元数据项相同，只不过它的重点放在了验证方面。默认提供程序会从校验 DataAnnotations 特性中读取元信息
TempData	ITempDataProvider	作为被放置在 TempData 集合中的数据存储提供服务。默认提供程序会使用会话状态
视图引擎	IViewEngine	作为能够解释视图模板和产生浏览器 HTML 标记的组件提供服务。默认视图引擎能将 ASPX 和 Razor 标记转译给 HTML

如你所见，这些并不是你每天所使用的组件。然而在我多年的编程生涯里，对其中的大部分至少自定义过一次，但是通常没有一次就自定义超过一个或两个的情况。

2. 一个现实的场景：替换 TempData 存储

TempData 字典用来存储从逻辑上讲属于当前请求的数据，但必须要跨越一个重定向来使用。第 4 章阐述了将 Post-Redirect-Get(PRG)模式上下文中的 TempData 字典用于输入表单的内容。根据 PRG 模式，应该用一个 GET 重定向到显示结果的视图来终止一个 POST 请求。整个请求(比如，验证消息)状态可能会在重定向过程中丢失。TempData 字典提供了一个用于任何数据(主要是 ModelState)所需的临时存储。

默认情况下，TempData 字典会将其内容保存到会话状态。直接使用 Session 与借助 TempData 的主要区别是，任何存储在 TempData 中的数据都会在连续请求终止后自动清除。换句话说，数据驻留在内存中只会用于两个请求:当前请求和下一个重定向。归根结底，TempData 带给内存的压力更小。

如果应用程序不允许使用会话状态呢？

要么你根据查询字符串给出完全不同的解决方案，或者直接为 TempData 内容提供不同的存储。那么什么才是不同的存储呢？它可以是一个 cookie(如果数据的总大小匹配对 cookie 的大小限制)，也可以是一个(分布式)缓存。如果选择后者，那么在每个用户的基础上存储数据就完全是你的责任了。自定义 TempData 提供程序看上去如下所示：

```
public class CookieTempDataProvider : ITempDataProvider
{
  protected override IDictionary<String, Object> LoadTempData
                                    (ControllerContext controllerContext)
  { ... }
  protected override void SaveTempData(ControllerContext controllerContext,
                                    IDictionary<String, Object> values)
  { ... }
}
```

可以访问 http://www.stackoverflow.com，通过直接检查 StackOverflow 问题找到一个此类的运行示例(以及在表 8-1 中列出的很多其他替换组件)。下面是一个我特别喜欢的示例：http://brockallen.com/2012/06/11/cookie-based-tempdata-provider。

3. 在应用程序中使用自定义组件

很长时间以来，注册自定义组件并没有标准方法。表 8-1 中所列出的每个组件都需要自己的 API 来集成到用户应用程序。你即将会看到，你现在有了一个额外的基于依赖性解析器的模型，它的大部分都类似于 ASP.NET MVC 早期版本所支持的自定义模型。

表 8-2 描述了如何注册表 8-1 中所列出的特殊的可替换组件。

表 8-2　注册可替换组件

提 供 程 序	描　　述

操作调用程序	// 在控制器类的构造函数中
	this.ActionInvoker = new YourActionInvoker();

控制器工厂	// 在 global.asax 的 Application_Start 中
	var factory = new YourControllerFactory();
	ControllerBuilder.Current.SetControllerFactory(factory);

字典值	// 在 global.asax 的 Application_Start 中
	var providerFactory = new YourValueProviderFactory();
	ValueProviderFactories.Factories.Add(providerFactory);

模型绑定器	// 在 global.asax 的 Application_Start 中
	ModelBinders.Binders.Add(typeof(YourType), new YourTypeBinder());

模型绑定器提供程序	// 在 global.asax 的 Application_Start 中
	var provider = new YourModelBinderProvider();
	ModelBinderProviders.BinderProviders.Add(provider);

模型元数据	//在 global.asax 的 Application_Start 中
	ModelMetadataProviders.Current = new YourModelMetadataProvider();

模型校验器	//在 global.asax 的 Application_Start 中
	var validator = new YourModelValidatorProvider();
	ModelValidatorProviders.Providers.Add(validator);

TempData	//在控制器类的构造函数中
	this.TempDataProvider = new YourTempDataProvider();

视图引擎	// 在 global.asax 的 Application_Start 中
	ViewEngines.Engines.Clear();
	ViewEngines.Engines.Add(new YourViewEngine());

我想再多谈一谈值提供程序。就像表 8-2 中所示的代码片段一样，你管理的不是一个值提供程序的集合，而是一个值提供程序工厂的集合。这就是说，对于每一个你打算添加的自定义值提供程序，你都要编写两个不同的类：值提供程序及其工厂。但是，仅工厂类应当在系统启动时注册。值提供程序工厂是一个精简封装，如下所示：

```
public class HttpCookieValueProviderFactory : ValueProviderFactory
{
  public override IValueProvider GetValueProvider
                                   (ControllerContext controllerContext)
```

```
    {
      return new HttpCookieValueProvider(controllerContext);
    }
}
```

这个类表示一个自定义值提供程序的工厂；在这方面，还有相当多的由开发者社区所提供的实现。可以在 StackOverflow 上找到一些。

8.1.2　服务定位器模式

服务定位器是在松散耦合系统设计中使用的流行模式。该模式的核心是一个全局的可访问工厂类，负责为那些执行指定协议的类实例提供服务。服务定位器与其客户端之间发生的基本交互可以概括为以这样开始的会话：我需要一个与这个类型作用一样的对象；你是否知道一些可以为我实例化的具体类型？之后服务定位器会通过返回一个像这样的实例或 null 作为回复。

“服务定位器”(以及像“服务定位”这样的表述)这个术语通常用于表示某个类获取外部依赖性的场景，因此要在扩展的时候保持开放，修改的时候保持关闭。那么服务定位器(或位置)模式与依赖注入之间的真正区别是什么呢？

1. 服务定位器与依赖注入对比

从功能上讲，服务定位器(SL)模式与依赖注入(DI)几乎完全相同，两者都是依赖反转原则——流行的 SOLID 中的“D”(SOLID 代表的是单一职责、开闭原则、里氏替换、接口隔离以及依赖反转原则)的具体实现。SL 历史上是构建松散耦合的可测试系统所广泛采用的第一种模式。后来 DI 作为完成同样任务的更好一些的方式被引入了。SL 和 DI 的区别多少有些模糊，取决于你从何种抽象程度来看待它们。当然，在源代码水平还有差异。使用 SL，类可以通过查询外部组件获取其依赖性。使用 DI，类可以通过构造函数(或公共属性)获得依赖性。

下面是使用 SL 的类的示例：

```
public class FinanceInfoService
{
  private IFinder _finder;
  private IRenderer _renderer;

  public FinanceInfoService()
  {
     _finder = ServiceLocator.GetService<IFinder>();
    _renderer = ServiceLocator.GetService<IRenderer>();
  }

  public String GetQuotesAsHtml(String symbols)
```

```
    {
      var stocks = _finder.FindQuoteInfo(symbols);
      return _renderer.RenderQuoteInfo(stocks);
    }
}
```

使用 DI 的相同类被重构后看起来像这样：

```
public class FinanceInfoService
{
  private IFinder _finder;
  private IRenderer _renderer;

  public FinanceInfoService(IFinder f, IRenderer r)
  {
      _finder = f;
      _renderer = r;
  }

  public string GetQuotesAsHtml(string symbols)
  {
    var stocks = _finder.FindQuoteInfo(symbols);
    return _renderer.RenderQuoteInfo(stocks);
  }
}
```

不同之处都在构造函数以及如何检索依赖性中。应该使用哪种方法呢？

对于从头开始构建的新系统，DI 是该优先考虑的；它会使代码保持得更整洁，更便于阅读和测试。使用 DI，依赖性在类的签名中是显式的。不过，周边框架要对依赖性的准备和注入负完全责任。对于现有系统，SL 更容易接入使用，因为它只会要求你将调用封装在一个公开接口的黑盒中。

2. ASP.NET MVC 中的 SL

ASP.NET MVC 设计有几个扩展点，但缺乏对 DI 的广泛支持。表 8-2 中列出的各种 API 证明了这一表述的可信度。要让现有框架通过添加新的扩展点得以更松散地耦合，服务定位器可能是最有效的，因为它是侵入性最小的解决方案。服务定位器会充当一个你在特定的点上安装的黑盒，并随后让它找出需要哪些协议以及如何得到这些协议。

与表 8-2 中所列示的 API 类似，ASP.NET MVC 为你提供了注册自己的服务定位器的能力，框架会在任何需要解析依赖性的时候使用。服务定位器是可选的，使用它或继续采用表 8-2 中所示的 API 在功能上是等效的。

当解析一个依赖性时，ASP.NET MVC 通常会首先使用内部服务定位器。依赖性能够被

单独注册还是多次注册，对实际行为的改变不大。但是从开发人员的角度来看，重要的是如果自定义组件以两种方式注册(使用服务定位器或表 8-2 中所示的任意 API)，则优先考虑服务定位器。

注意：

单独注册的组件对应用程序来说一定是唯一的一个组件。一个例子就是控制器工厂——你不能在同一应用程序中拥有两个工厂。多次注册的组件的例子有视图引擎和模型验证提供程序。在这两种情况下，可以注册多个实例并向系统的其他部分公开。

3. 定义依赖性解析器

ASP.NET MVC 服务定位器的实现是由对称为依赖性解析器的用户定义对象的简单封装所构成的。依赖性解析器——对每个应用程序来说是唯一的——是一个实现下列接口的对象：

```
public interface IDependencyResolver
{
  Object GetService(Type serviceType);
  IEnumerable<Object>GetServices(Type serviceType);
}
```

在解析器中设置什么样的逻辑完全取决于你。它可以简单到一个 switch 语句，检查类型并返回一个新创建的固定类型的实例。它也可以复杂到从配置文件中读取信息和使用反射来创建实例。最后，它可以基于 Unity 或其他任何 IoC 框架。下面是一个简单实用的解析器：

```
public class SampleDependencyResolver : IDependencyResolver
{
    public object GetService(Type serviceType)
    {
        if (serviceType == typeof(ISomeClass))
            return new SomeClass();
        ...
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return Enumerable.Empty<Object>();
    }
}
```

接下来显示的代码是一个使用 Unity(及其配置部分)来解决依赖性的解析器示例：

```
Public class UnityDependencyResolver : IDependencyResolver
{
```

```
    private readonly IUnityContainer _container;

    public UnityDependencyResolver() : this(new UnityContainer().
                                         LoadConfiguration())
    {
    }
    public UnityDependencyResolver(IUnityContainer container)
    {
      _container = container;
    }

    public Object GetService(Type serviceType)
    {
      return _container.Resolve(serviceType);
    }

    public IEnumerable<Object> GetServices(Type serviceType)
    {
      return _container.ResolveAll(serviceType);
    }
  }
```

使用 ASP.NET MVC 框架通过 DependencyResolver 类的 SetResolver 方法注册自己的解析器，如下所示：

```
protected void Application_Start()
{
   // Prepare and configure the IoC container
   var container = new UnityContainer();
   ...

   // Create and register the resolver
   var resolver = new UnityDependencyResolver(container);
   DependencyResolver.SetResolver(resolver);
}
```

如果在解析器的内部使用 IoC 框架，就需要明白，最好的方式是向其提供注册类型的列表。如果喜欢通过连贯代码传递该信息，则需要在创建解析器之前完全配置 IoC 容器对象。如果打算通过使用 web.config 文件配置 IoC，则可以使用解析器的默认构造函数——就 Unity 而言——它包含一个加载配置数据的调用。不过请注意，如果针对的是一个不同的 IoC 框架，则可能需要更改此代码。

重要提示：

依赖性解析器是一个内部工具，开发人员可以选择性地使用来创建自己的自定义组件而不是系统组件。依赖性解析器的功能仅限于 ASP.NET MVC 所支持的方式使用。换句话说，例如，如果 ASP.NET MVC 不在创建控制器缓存之前调用解析器，你想靠自己去替换内置缓存就非常难了。

8.2 在控制器中添加特性

在 ASP.NET MVC 中你会发现一种完全不同的基于可以附加到控制器类和方法上的特性的自定义形式。这些特性通常称为操作筛选器。操作筛选器是一段围绕操作方法的执行而运行的代码，并可用于修改和扩展在方法本身中硬编码的行为。

8.2.1 操作筛选器

操作筛选器可完全由下面的接口表示：

```
public interface IActionFilter
{
  void OnActionExecuting(ActionExecutingContext filterContext);
  void OnActionExecuted(ActionExecutedContext filterContext);
}
```

可以看到，它为你提供了一个可以在操作执行之前和之后运行代码的挂钩。从筛选器内部，可以访问请求和控制器上下文，还可以读取和修改参数。

1. 嵌入式和外部筛选器

用户定义的每个控制器都是从 Controller 类继承而来的。这个类作为可重写的受保护成员实现了 IActionFilter 并公开了 OnActionExecuting 和 OnActionExecuted 方法。这意味着每个控制器类都让你有机会决定之前做什么、之后做什么、或者在调用指定方法之前和之后做什么。让我们看一些在调用 Index 方法时会添加特设响应标头的代码。

```
protected DateTime StartTime;
protected override void OnActionExecuting(ActionExecutingContext
    filterContext)
{
    var action = filterContext.ActionDescriptor.ActionName;
    if (String.Equals(action, "index",
       StringComparison.CurrentCultureIgnoreCase))
    {
```

```
        StartTime = DateTime.Now;
    }

    base.OnActionExecuting(filterContext);
}
protected override void OnActionExecuted(ActionExecutedContext filterContext)
{
var action = filterContext.ActionDescriptor.ActionName;
if (String.Equals(action, "index", StringComparison. CurrentCultureIgnoreCase))
{
var timeSpan = DateTime.Now - StartTime;
filterContext.RequestContext.HttpContext.Response.AddHeader(
"despos-mvc4", timeSpan.Milliseconds.ToString());
}

base.OnActionExecuted(filterContext);
}
```

图 8-1 显示了该方法计算其执行的毫秒数，并将该数值写入新的响应标头。

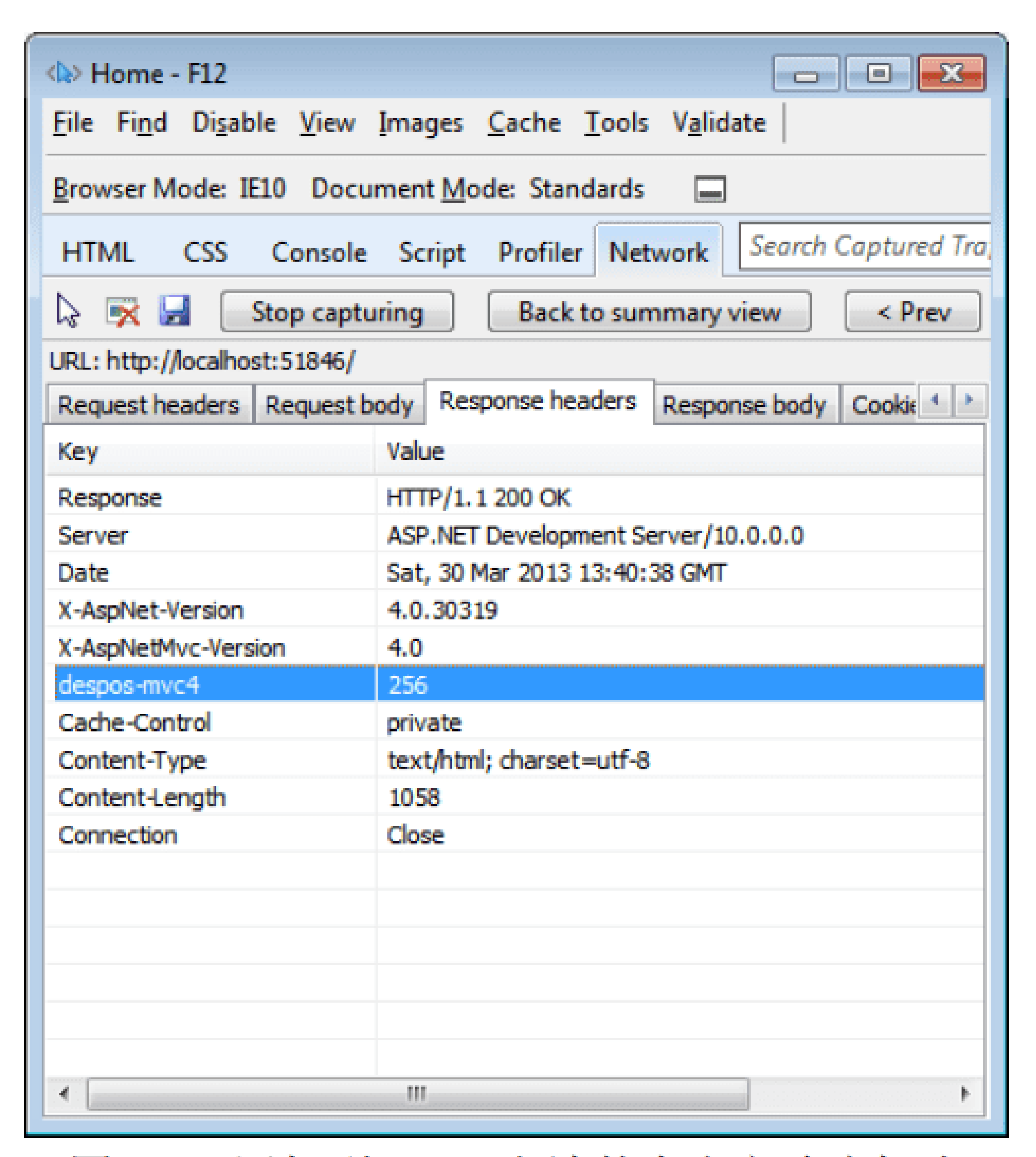


图 8-1　添加到 Index 方法的自定义响应标头

如果在一个控制器类中重写这些方法，最后所有的控制器方法都会有一个 before/after 代码的单独存储库。只要你想要为每个方法执行相同的操作，这种方式都管用。如果要在每个方法的基础上区分操作，把操作筛选器用作特性可能是一个值得尝试的技巧。

操作筛选器作为一个特性被实现，它提供了将某些行为附加到控制器操作方法上的声明

式途径。通过编写操作筛选器，可以挂接操作方法的执行管道，并使它适应于你的需要。用这种方式，还可将并不严格属于控制器的逻辑从控制器类中去除。这样的话，这种特殊行为便具有了可重用性，更重要的是，具备了可选性。操作筛选器对于实现影响你控制器生存期的横切关注点的解决方案来说是不错的。

2. 操作筛选器的分类

操作筛选器根据其实际完成的任务分成不同的类型。操作筛选器的特点是具有一个接口；每种类型的筛选器具有一个不同的接口。特殊的操作筛选器包括异常筛选器、授权筛选器和结果筛选器。表 8-3 列出了 ASP.NET MVC 中的操作筛选器类型。

表 8-3　ASP.NET MVC 中操作筛选器的类型

筛选器接口	描　　述
IActionFilter	定义两个方法，一个在控制器操作之前执行，另一个在其之后执行
IAuthenticationFilter	定义在操作管道中早期执行的方法，让你可以自定义身份验证处理过程并且还将不同的身份验证策略用于不同的操作方法和控制器上
IAuthorizationFilter	定义在身份验证之后执行的方法，让你可以验证用户是否有权执行操作
IExceptionFilter	定义在控制器操作执行期间抛出异常时运行的方法
IResultFilter	定义两个方法，一个在操作结果处理过程之前执行，另一个在其之后执行

表 8-3 中所有接口的实现会产生在 Controller 类上的几个附加方法。表 8-4 列示了这些方法以及对其的注释。

表 8-4　控制器类中的筛选器方法

方　　法	描　　述
OnActionExecuting	在执行操作方法前调用
OnActionExecuted	在操作方法执行完成后调用
OnAuthentication	当操作方法的身份验证发生时调用
OnAuthorization	当授权操作方法的身份验证时调用
OnException	当操作方法中发生异常时调用
OnResultExecuting	在操作结果被执行前调用
OnResultExecuted	在操作结果执行完成后调用

所有这些方法都是虚拟的且受保护的，可以在你的控制器类中重写它们以实现更具体的行为。

让我们进一步研究一些预定义的操作筛选器。

3. 内置的操作筛选器

ASP.NET MVC 附带了一些预定义的筛选器，其中的一些在前面几章中讨论过了：HandleError、Authorize 和 OutputCache 就是其中的一部分。表 8-5 列出了 ASP.NET MVC 中可用的内置筛选器。

表 8-5　ASP.NET MVC 中预定义的筛选器

筛　选　器	描　　述
AsyncTimeout	将一个操作方法标记为将异步执行并在指定毫秒值内终结的方法。异步方法还具有一个伴随特性用于设置不超时。该伴随特性是 NoAsyncTimeout
Authorize	将一个操作方法标记为仅能由指定用户、角色或两者一起访问的方法
ChildActionOnly	将一个操作方法标记为仅能作为子操作在呈现操作运行期间执行的方法
HandleError	将一个操作方法标记为需要对该方法执行期间抛出的异常自动处理的方法
OutputCache	将一个操作方法标记为需要缓存其输出的方法
RequireHttps	将一个操作方法标记为需要安全请求的方法。如果该方法通过 HTTP 调用，则特性会强制通过 HTTPS 连接重定向到相同的 URL，如果可能的话
ValidateAntiForgeryToken	将一个操作方法标记为需要针对每个 POST 请求页面中防伪令牌进行验证的方法
ValidateInput	将一个操作方法标记为其提交的输入数据可能(或可能不)需要验证的方法

每个控制器方法可以用多个筛选器来修饰。因此处理筛选器的顺序就很重要了。表 8-5 中所列的所有特性均从基类 FilterAttribute 派生而来，它定义了一个基本属性 Order。Order 属性表明了多个特性的应用顺序。默认情况下，Order 属性会被分配一个–1 值，表示未指定顺序。但是，任何未指定顺序的筛选器始终会在具有固定顺序的筛选器之前执行。最后，注意如果对一个方法上的两个或多个操作筛选器显式设置相同的顺序，将引发异常。

4. 全局筛选器

可以把筛选器应用于单个方法，也可以应用于整个控制器类。如果将筛选器应用于控制器类，则会对所有由控制器公开的操作方法产生影响。与此相反的是，全局过滤器是在应用程序启动时注册并自动应用于任何控制器类的任何操作。

默认情况下，HandleError 筛选器是在 global.asax 中全局注册的，这意味着它对所有操作方法提供了一些异常处理功能。全局筛选器就是以另外一种方式注册的普通操作筛选器，如下所示：

```
GlobalFilters.Filters.Add(new HandleError());
```

GlobalFilters 集合在各个操作被调用之前会由当前的操作调用程序对其进行检查，并且

找到的所有筛选器都会被添加到已启用的筛选器列表中，以对操作进行预处理和后处理。我还会在本章稍后继续谈及操作调用器和用于每个操作的筛选器列表。

8.2.2　操作筛选器库

总体来看，操作筛选器在 ASP.NET MVC 内部形成了一个嵌入式的面向方面的框架。ASP.NET MVC 提供了许多预定义的筛选器，但也可以自己编写。编写操作筛选器的时候，你通常要继承 FilterAttribute，然后实现一个或多个在表 8-3 中所定义的接口。不过，大部分的时间，可以采用较短的路径——派生自 ActionFilterAttribute。ActionFilterAttribute 类是用于创建自定义操作筛选器的另一个更丰富的基类。它继承自 FilterAttribute，并且提供了一个用于表 8-3 中列出的所有接口的默认实现。让我们看几个操作筛选器示例是怎么编写的。

注意：

操作筛选器是封装特定行为的自定义组件。当你想隔离这种行为并希望轻易复制它时，就可以编写一个操作筛选器。行为的可重用性是决定是否要编写操作筛选器的因素之一，但不是唯一的因素。操作筛选器也起到保持控制器代码至精至简的作用。一般而言，每当你的控制器方法代码用分支代码和条件语句填充时，应该停下来想一想是否某些分支代码(或重复代码)可以移到操作筛选器。这会在很大程度上提高代码的可读性。

1. 添加响应标头

拥有一个自定义操作筛选器的经典方案是将你希望应用于许多(但不一定是全部)操作方法的重复行为封装起来。一个典型的例子是在方法的响应中添加一个自定义标头。在本章的前面部分，你看到了如何通过在控制器类中使用 IActionFilter 本地实现来达成这一点。下面的代码显示了如何将代码移出控制器使其变成一个单独的类：

```
public class AddHeaderAttribute : ActionFilterAttribute
{
    public String Name { get; set; }
    public String Value { get; set; }

    public override void OnActionExecuted(ActionExecutedContext
        filterContext)
    {
        if (!String.IsNullOrEmpty(Name) && !String.IsNullOrEmpty(Value))
            filterContext.RequestContext.HttpContext.Response.AddHeader
                (Name, Value);
        return;
    }
}
```

现在你有了一段易于管理的代码。可以将其附加到任意数量的控制器操作、一个控制器的所有操作，甚至于全局到所有控制器。所有这些只需要添加一个特性，如下所示：

```
[AddHeader(Name="Action", Value="About")]
public ActionResult About()
{ ... }
```

你可能认为此示例并不完全等同于之前的那个，在其中我们计算出了操作的起始时间和完成时间之间的间隔。事实上，AddHeader 筛选器只是添加了一个固定标头。可以派生一个新类——比如 ReportDuration——并在其中应用你需要的所有逻辑：

```
public class ReportDurationAttribute : AddHeaderAttribute
{
    protected DateTime StartTime;

    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        StartTime = DateTime.Now;
        base.OnActionExecuting(filterContext);
    }

    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        var timeSpan = DateTime.Now - StartTime;
        Value = timeSpan.Milliseconds.ToString();
        base.OnActionExecuted(filterContext)
    }
}
```

我们看一个稍复杂点儿的例子。在这个例子中，我们将压缩方法响应，这是一个有用的功能，尤其是当返回大的 HTML 或二进制数据块的时候。

2. 压缩响应

近来，HTTP 压缩成为了一种几乎每个网站都承受得起的功能，因为在这方面有问题的浏览器数目几近于零(在过去 10 年间发布的任何浏览器都能够识别这个最流行的压缩方案)。

在 ASP.NET Web Forms 中，压缩一般是通过 HTTP 模块拦截请求和压缩响应来实现的。你还可以在互联网信息服务(IIS)级别开启压缩。这两个选项在 ASP.NET MVC 中都好用，决定使用哪一个选项取决于你。你通常需要根据所要控制的参数来做决定，包括要压缩资源的 MIME 类型、压缩级别、要压缩的文件等。

ASP.NET MVC 提供了特别容易实现的第三个选项：一个对压缩事项进行设置的特定操作筛选器。用这种方式，可以在不必编写 HTTP 模块的情况下控制某种特定的 URL。我们来

看另一个在用于特定方法的响应流中添加压缩的操作筛选器示例吧。

一般情况下，HTTP 压缩由两个参数控制：浏览器随每个请求发送的 Accept-Encoding 标头和 Web 服务器随每个响应发送的 Content-Encoding 标头。Accept-Encoding 标头指明浏览器只能处理指定的编码——通常情况下是 gzip 和 deflate[①]。Content-Encoding 标头会指明响应的压缩格式。请注意 Accept-Encoding 标头只是一个由浏览器发送的请求标头；服务器并没有必要返回压缩的内容。

谈到编写压缩过滤器，最难的部分在于获得对浏览器所请求内容的充分理解。下面是一些有用的代码：

```
public class CompressAttribute : ActionFilterAttribute
{
   public override void OnActionExecuting(ActionExecutingContext
      filterContext)
   {
      // Analyze the list of acceptable encodings
      var preferredEncoding = GetPreferredEncoding(filterContext.
                              HttpContext.Request);

      // Compress the response accordingly
      var response = filterContext.HttpContext.Response;
      response.AppendHeader("Content-encoding",
                              preferredEncoding.ToString());

      if (preferredEncoding == CompressionScheme.Gzip)
         response.Filter = new GZipStream(response.Filter,
                              CompressionMode.Compress);
      if (preferredEncoding == CompressionScheme.Deflate)
         response.Filter = new DeflateStream(response.Filter,
                              CompressionMode.Compress);

      return;
   }

   private CompressionScheme GetPreferredEncoding(HttpRequestBase request)
```

① gzip 最早由 Jean-Loup Gailly 和 Mark Adler 创建，用于 UNIX 系统的文件压缩。我们在 Linux 中经常会用到后缀为.gz 的文件，它们就是 gzip 格式的。现今已经成为互联网上使用非常普遍的一种数据压缩格式，或者说一种文件格式。HTTP 协议上的 gzip 编码是一种用来改进 Web 应用程序性能的技术。大流量的 Web 站点常常使用 gzip 压缩技术来让用户感受更快的速度。

deflate 是同时使用了 LZ77 算法与哈夫曼编码(Huffman Coding)的一个无损数据压缩算法。它最初是由 Phil Katz 为他的 PKZIP 归档工具第二版所定义的，后来定义在 RFC 1951 规范中。

```
    {
        var acceptableEncoding = request.Headers ["Accept-Encoding"].ToLower();

        if (acceptableEncoding.Contains("gzip"))
            return CompressionScheme.Gzip;
        if (acceptableEncoding.Contains("deflate"))
            return CompressionScheme.Deflate;

        return CompressionScheme.Identity;
    }

    enum CompressionScheme
    {
        Gzip = 0,
        Deflate = 1,
        Identity = 2
    }
}
```

将 Compress 特性应用于如下所示的方法：

```
[Compress]
public ActionResult Index()
{ ... }
```

图 8-2 显示出 Content-Encoding 响应标头设置正确，而响应被理解并在浏览器中解压缩。

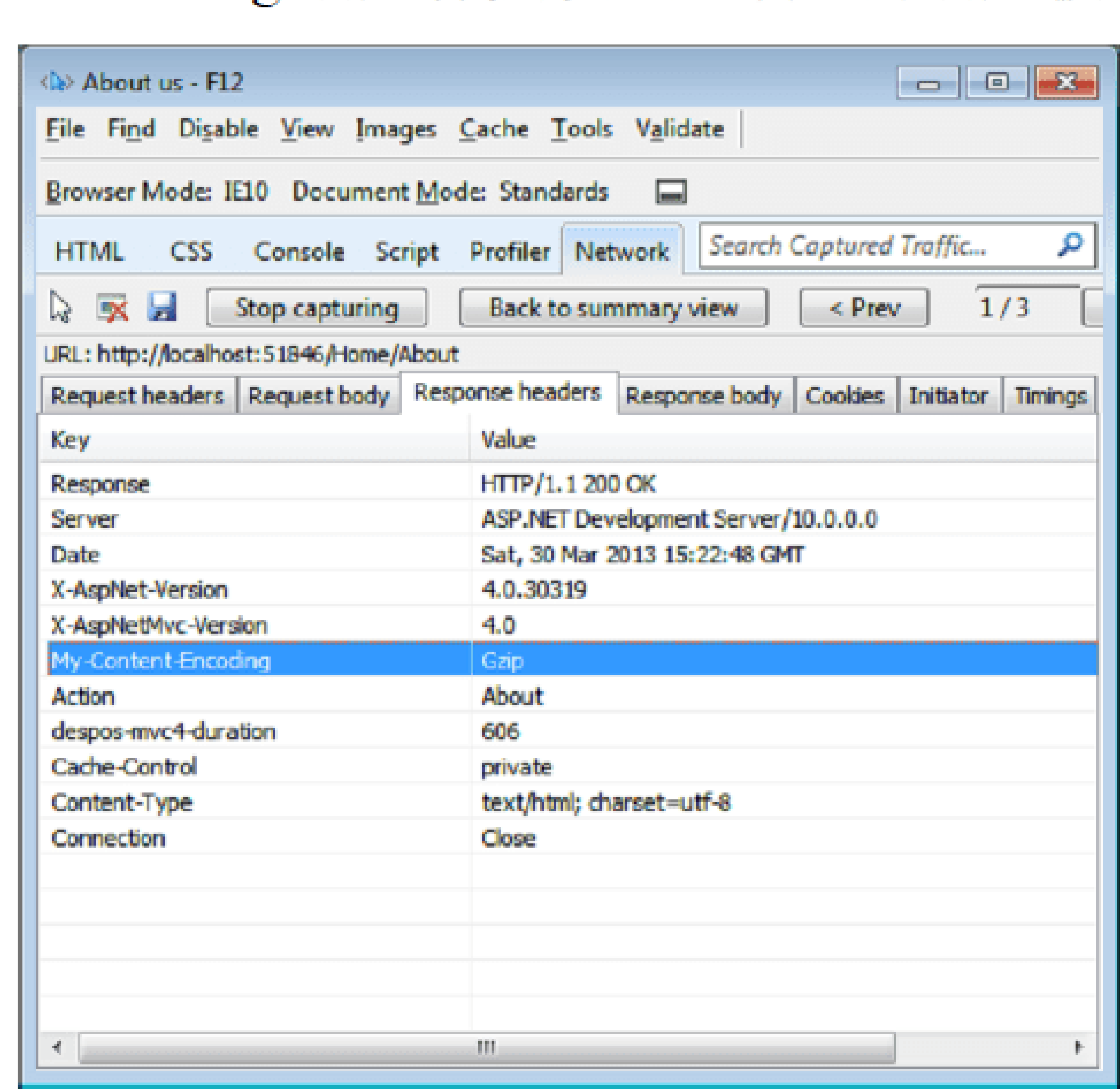


图 8-2　底层 HTTP 流量的检视显示了用到的带有编码描述的自定义标头

几乎所有浏览器都将 Accept-Encoding 标头设置为字符串“gzip,deflate”，但这并不是唯

一可行的。可以在 RFC 2616 (参见 http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html)中看到，Accept 标头字段支持将 q 参数作为对可接受值分配优先权的一种方式。下面的字符串对于编码是可接受的值：

```
gzip, deflate
gzip;q=.7,deflate
gzip;q=.5,deflate;q=.5,identity
```

即使 gzip 出现在所有的字符串中，但也只有出现在第一个字符串中它才是首选。如果未指定值，q 参数即设置为 1；这是将 deflate 分配到第二个字符串中，并为第三个字符串中的身份分配一个比 gzip 高的级别。所以，只检查 gzip 是否出现在编码字符串中仍然会回发一些浏览器可接受的内容，但并不会充分考虑浏览器的偏好。要编写一个将通过 q 参数所表示的优先权(如果有的话)纳入考虑的 Compress 特性，你需要完善 GetPreferredEncoding 方法，如下所示：

```
private CompressionScheme GetPreferredEncoding(HttpRequestBase request)
{
   var acceptableEncoding = request.Headers["Accept-Encoding"].ToLower();
   acceptableEncoding = SortEncodings(acceptableEncoding);

   if (acceptableEncoding.Contains("gzip"))
      return CompressionScheme.Gzip;
   if (acceptableEncoding.Contains("deflate"))
      return CompressionScheme.Deflate;

   return CompressionScheme.Identity;
}
```

SortEncodings 方法会解析标头字符串并提取对应于最高优先权选择的那一段。

3. 视图选择器

第 5 章“ASP.NET MVC 应用程序特性”中讨论了 CultureAttribute 筛选器，用来将特定的区域强制用于指定的操作方法，从而获得为请求启用本地化的效果。CultureAttribute 筛选器如果是作为全局筛选器安装的，就会正常运行。我不会再讨论操作筛选器实现本地化的用途了。然而，接下来要说的筛选器，仍然属于这个筛选器类，可以为指定的操作选择不同的视图。假设你要根据请求浏览器的功能切换到一个不同的视图模板。

注意：

虽然从了解操作筛选器在 ASP.NET MVC 中使用范围的角度来说，下面的示例仍然有用，但示例揭示的特定功能可以通过使用一种名为显示模式的新功能在最新的 ASP.NET MVC 中

以更有效和更紧凑的方式得以实现。我将在第 13 章“构建用于多种设备的网站”中详细讨论显示模式。

创建一个多任务的应用程序不容易，但更重要的是，它往往是一个特定于项目的解决方案。接下来我将说明筛选器在帮助你筹划多任务解决方案中可以发挥的作用。说得更清楚一些，多任务应用程序是可以根据请求的浏览器不同而以不同的方式服务相同请求的一种应用程序。

直到几年前，一个经典的多任务示例就是使用用于富浏览器的页面和用于欠缺丰富性的桌面浏览器的页面。今天，Ajax 和 JavaScript 库已经明显弱化了这一问题的影响，因为 JavaScript 库(比如 jQuery)现在可以对开发人员隐藏大部分的差异。然而，新兴的分化在于桌面浏览器和移动设备浏览器之间。这两个类别的界限不像看起来那样清晰。归根结底，界限定义取决于你。定义界限意味着你要决定如何处理智能手机和平板电脑，并决定使用功能较弱的嵌入到移动设备和移动电话的浏览器要完成什么任务。然而，出于本章目标的考虑，我会假设你知道这两者之间的界限在哪里。所以，我会继续进行对知道如何根据用户代理字符串和你的规则切换视图的操作筛选器的讨论。

方法其实就是编写一个在操作执行以后介入且在操作调用程序开始之前处理结果的筛选器。根据表 8-3 中所示的分类，这从技术上讲是结果筛选器。让我们看看下面示例中的源代码：

```
public class BrowserSpecificAttribute : ActionFilterAttribute
{
   public override void OnResultExecuting(ResultExecutingContext filterContext)
   {
       ...
   }
}
```

该筛选器继承自 ActionFilterAttribute 并重写了 OnResultExecuting 方法。该方法在执行操作方法以后被调用，但会在操作结果被处理之前生成用于浏览器的响应。

```
public override void OnResultExecuting(ResultExecutingContext filterContext)
{
    var viewResult = filterContext.Result as ViewResult;
    if (viewResult == null)
        return;

    // Figure out the view name
    var context = filterContext.Controller.ControllerContext;
    var viewName = viewResult.ViewName;
    if (String.IsNullOrEmpty(viewName))
```

```
        viewName = context.RouteData.GetRequiredString("action");
    if (String.IsNullOrEmpty(viewName))
        return;

    // Resolve the view selector
    var viewSelector = DependencyResolver.Current.GetService(typeof
                        (IViewSelector)) as IViewSelector;
    if (viewSelector == null)
        viewSelector = new DefaultViewSelector();

    // Figure out the browser name
    var isMobileDevice = context.HttpContext.Request.Browser.IsMobileDevice;
    var browserName = (isMobileDevice ?"mobile" :context.HttpContext.
                      Request.Browser.Browser);

    // Get the name of the browser-specific view to use
    var newViewName = viewSelector.GetViewName(viewName, browserName);
    if (String.IsNullOrEmpty(newViewName))
        return;

    // Is there such a view?
    var result = System.Web.Mvc.ViewEngines.Engines.FindView(context,
                newViewName, viewResult.MasterName);
    if (result.View != null)
        viewResult.ViewName = newViewName;
}
```

采用的算法很简单。使用 ControllerContext 对象，该筛选器会从请求上下文检索 Request 对象；从那里可以查明当前浏览器的功能。浏览器的名称会被用作决定要选择的下一个视图的鉴别器。

IViewSelector 类型的对象会解析指定浏览器名称的视图名称。下面显示了一个默认的视图选择器的实现：

```
public class DefaultViewSelector : IViewSelector
{
    public String GetViewName(String viewName, String browserName)
    {
        return String.Format("{0}_{1}", viewName, browserName);
    }

    public String GetMasterName(String masterName, String browserName)
    {
        return String.Format("{0}_{1}", masterName, browserName);
```

```
    }
}
```

该代码假定，指定一个视图名称为 Index，对于互联网浏览器(IE)——该视图的特定版本名为 Index_IE，而 Firefox 的版本名为 Index_Firefox，依此类推。在筛选器确定了要显示的候选视图的名称以后，它还会检查当前的视图引擎，看看是否支持该视图。如果支持，ViewResult 要呈现的 ViewName 属性就会被设置成特定于浏览器的视图。如果没有发现特定于浏览器的视图，你也不需要做什么，因为由操作方法调用的通用视图仍然会起作用，如图 8-3 中所示。

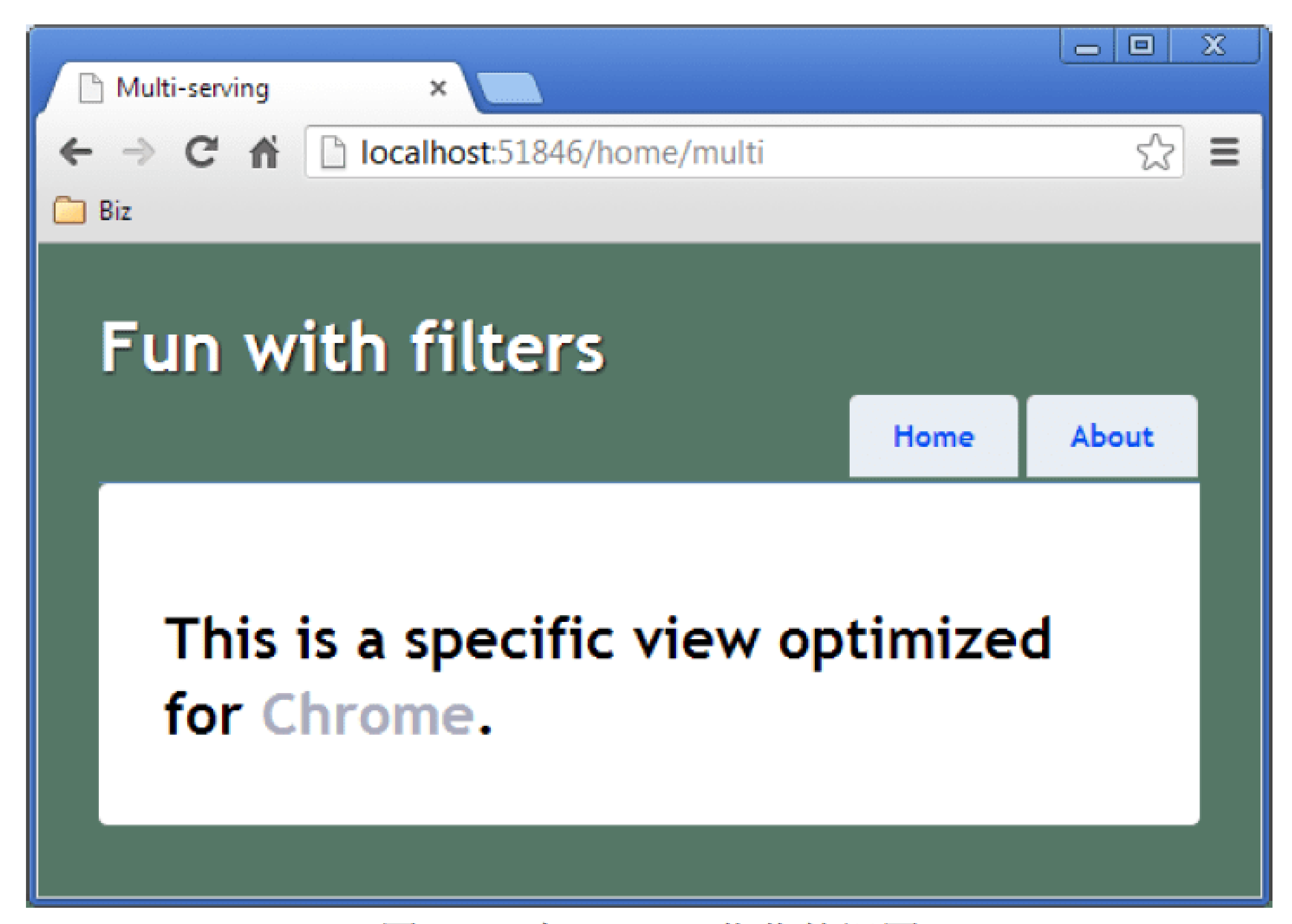


图 8-3　为 Chrome 优化的视图

如果请求浏览器刚好是移动设备的浏览器，则会选择名为 xxx_mobile 的视图。

使用该特性再简单不过了。你所需要做的只是用该特性来修饰控制器方法，如下所示：

```
[BrowserSpecific]
public virtual ActionResult Index()
{ ... }
```

这样一个操作筛选器使你不必向每个控制器方法添加一堆 if 语句，以便为每个受支持的浏览器返回一个不同的 ViewResult 对象：

```
public virtual ActionResult Index()
{
  if(GetCurrentBrowser() == "IE")
  return View("Index_IE");
  ...
}
```

你可能仍然认为这段代码在某些边缘情况下是需要的，但操作筛选器可以让你得以将其从控制器类分离出来，从而简化整个设计。

注意：

从这个例子中，可以清楚地看到为什么 ASP.NET MVC 有别于 Web Forms 以及它们之间相异的程度。切换视图在 Web Forms 中也绝对可能实现，但它需要一点非常规处理——切换母版页，以编程方式加载用户控件和以编程方式更改模板。这是在设计 Web Forms 时不会优先考虑的编程方面的内容，因为它涉及的问题不太敏感。现在不一样了，ASP.NET MVC 使它更易于处理与开发人员有关的问题。

8.2.3　特殊筛选器

到目前为止，操作筛选器被认为是旨在拦截操作方法执行的几个阶段的组件。如果想添加一些代码来帮助决定某指定方法是否适于服务某指定操作呢？对于这种自定义类型，就需要另一个类别的筛选器了：操作选择器。

操作选择器有两种类别：操作名称选择器和操作方法选择器。名称选择器决定它们修饰的方法是否可用于服务指定的操作名称。方法选择器决定具有匹配名称的方法是否可用于服务指定的操作。方法选择器通常会基于其他运行时条件给出响应。

1. 操作名称选择器

操作名称选择器的基类是 ActionNameSelectorAttribute。该类具有一个简单的结构，如下面的代码所示：

```
public abstract class ActionNameSelectorAttribute : Attribute
{
  public abstract Boolean IsValidName(ControllerContext controllerContext,
    String actionName, MethodInfo methodInfo);
}
```

选择器的目的很简单：检查指定的操作名称是否是用于方法的有效操作名称。

在 ASP.NET MVC 中，只有一个操作名称选择器：ActionName 特性，可用于对控制器方法命名别称。下面是一个示例：

```
[ActionName("Edit")]
public ActionResult EditViaPost(String customerId)
{
  ...
}
```

ActionName 特性的实现是很寻常的，如以下代码所示：

```
public sealed class ActionNameAttribute : ActionNameSelectorAttribute
{
    public ActionNameAttribute(String name)
    {
        if (String.IsNullOrEmpty(name))
            throw new ArgumentException();
        Name = name;
    }

    public override Boolean IsValidName(ControllerContext controllerContext,
                                        String actionName, MethodInfo methodInfo)
    {
        // Check that the action name matches the specified name
        return String.Equals(actionName, Name, StringComparison.
                             OrdinalIgnoreCase);
    }

    public String Name { get; set; }
}
```

该特性的整体作用是，它会在逻辑上将控制器方法重命名为它所适用的名称。例如，在前面的示例中，方法名称是 EditViaPost，但除非从路由过程中产生的操作名称为 Edit，否则不会调用它。

2. 操作方法选择器

操作方法选择器对开发人员来说是一个更强大、更具吸引力的工具。在系统寻找可以处理请求的控制器方法的初步阶段，方法选择器只会指明某指定方法是否有效。很明显，这类选择器会基于某些运行时条件确定其响应。下面是基类的定义：

```
public abstract class ActionMethodSelectorAttribute : Attribute
{
    public abstract Boolean IsValidForRequest(
        ControllerContext controllerContext, MethodInfo methodInfo);
}
```

在 ASP.NET MVC 中，存在着不少预定义的方法选择器。它们是 AcceptVerbs、NonAction，还有几个引入以简化编码(HttpDelete、HttpGet、HttpPost 和 HttpPut)的 HTTP 特定的选择器。让我们看看其中的一些。

NonAction 特性用来阻止修饰的方法处理当前操作。下面是其如何实现的代码：

```
public override Boolean IsValidForRequest(
    ControllerContext controllerContext, MethodInfo methodInfo)
```

```
{
   return false;
}
```

AcceptVerbs 特性会接收列出的作为参数的受支持 HTTP 动词，并对照列表检查当前的动词。下面是一些详细代码：

```
public override Boolean IsValidForRequest(
   ControllerContext controllerContext, MethodInfo methodInfo)
{
   if (controllerContext == null)
      throw new ArgumentNullException("controllerContext");

   // Get the (overridden) HTTP method
   var method = controllerContext.HttpContext.Request.
                GetHttpMethodOverride();

   // Verbs is an internal member of the AcceptVerbsAttribute class
   return Verbs.Contains<String>(method, StringComparer.
                OrdinalIgnoreCase);
}
```

注意 GetHttpMethodOverride 方法的使用是按照客户端的意图检索实际的动词。该方法会读取名为X-HTTP-Method-Override 的标头字段或参数中的值(关于 X-HTTP-Method-Override 的更多信息，请参见 http://code.google.com/apis/gdata/docs/2.0/basics. html#UpdatingEntry)。这是一个让浏览器放置 HTTP 动词的通用协议，即使物理请求是 GET 或 POST。该方法不是在 HttpRequest 对象上进行本地定义的，而是作为 HttpRequestBase 的一种扩展方法添加到 ASP.NET MVC 中的。

其他选择器是直接根据 AcceptVerbs 实现的，如下所示的用于 HttpPost 的代码：

```
public sealed class HttpPostAttribute : ActionMethodSelectorAttribute
{
   private static readonly AcceptVerbsAttribute _innerAttribute;

   public override bool IsValidForRequest(
       ControllerContext controllerContext, MethodInfo methodInfo)
   {
     return _innerAttribute.IsValidForRequest(controllerContext,
       methodInfo);
   }
}
```

我们看看如何编写一个自定义的方法选择器。

3. 将方法仅限于 Ajax 调用

你所需要的只是一个从 ActionMethodSelectorAttribute 继承的类，并重写 IsValidForRequest 方法：

```
public class AjaxOnlyAttribute : ActionMethodSelectorAttribute
{
    public override Boolean IsValidForRequest(
            ControllerContext controllerContext, MethodInfo methodInfo)
    {
        return controllerContext.HttpContext.Request.IsAjaxRequest();
    }
}
```

任何以此特性标记的方法只有在服务于通过浏览器的 XMLHttpRequest 对象所放置的调用时才会启用。

```
[AjaxOnly]
public ActionResult Details(Int32 customerId)
{
    var model = ...;
    return PartialView(model);
}
```

如果尝试根据路由调用一个 URL，必须要映射到一个 Ajax 方法，才不会出现异常。

4. 将方法限于指定的提交按钮

在 Web Forms 中，每个页面都有一个单独的 HTML 表单，而几乎每一个表单都包含多个提交按钮。然而，每个按钮有其自身的 click 处理程序，用于确定在单击时要采取的操作。而在 ASP.NET MVC 中，可以随你所愿使用无数个 HTML 表单，每一个都会提交到不同的固定 URL。既然每个 HTML 表单有两个以上的提交按钮，你如何确定是哪一个按钮被单击了？

在 HTML 级别，提交的表单会上载输入字段的内容以及一个单击按钮固有的名称/值对。名称标记指的是按钮的名称特性；值标记指的是按钮的值特性。可惜，按钮的值特性是说明文字。你不能通过检查说明文字可靠地找出单击了哪个按钮。至少，这要受本地化影响，或者如果使用了图片按钮的话，这些就都可以跳过了。

在 ASP.NET MVC 中解决该问题的最简单办法是在表单中添加一点 javascript 代码，这样在单击每个按钮时，它便会更改表单的操作特性以反映其名称。下面是一个简单的带有两个输入字段和两个提交按钮的 HTML 表单，该表单用于更新或删除一些内容：

```
<form name="myForm" id="myForm" method="post">
   <input type="text" />
   <input type="text" />
   <hr />
   <input type="submit" value="Update" name="updateAction"
          onclick="setAction('update')" />
   <input type="submit" value="Delete" name="deleteAction"
          onclick="setAction('delete')" />
</form>
```

名为 setAction 的 JavaScript 函数执行以下操作：

```
<script type="text/javascript">
    function setAction(action) {
        document.getElementById("myForm").action = action;
    }
</script>
```

这招很管用，但烦人的是，你必须在每个 ASP.NET MVC 应用程序中为你编写的每个表单重复添加此代码。一个特设的方法选择器(以及内置的 ActionName 特性)更适合用来完成这一工作：

```
public class OnlyIfPostedFromButtonAttribute : ActionMethodSelectorAttribute
{
    public String SubmitButton { get; set; }
    public override Boolean IsValidForRequest(ControllerContext
controllerContext,
                                              MethodInfo methodInfo)
    {
        // Check if this request is coming through the specified submit button
        var o = controllerContext.HttpContext.Request[SubmitButton];
        return o != null;
    }
}
```

选择器会公开一个公共属性，通过它可以指明与方法关联的按钮名称。在估算时，选择器只会在被提交的表单数据集中寻找其伴随按钮的名称。根据 HTML 的设计，只有当表单是通过单击那个按钮提交的情况下才能找到匹配。让我们看看在下面的控制器代码中如何使用此选择器：

```
[HttpPost]
[ActionName("Demo")]
[OnlyIfPostedFromButton(SubmitButton = "updateAction")]
public ActionResult Demo_Update()
{
```

```
    ViewData["ActionName"] = "Update";
    return View("demo");
}
```

该代码解析如下：调用程序只有在请求是 POST、操作名称是 demo，并且表单是通过使用名为 updateAction 的提交按钮来提交的，才可以选择这种方法(Demo_Update)。

8.2.4　构建动态的加载筛选器

操作筛选器无疑是一个强大的机制，开发人员可用来决定到底如何执行一个指定的操作方法。但是，从你目前所看到的而言，操作筛选器依然是一种静态的机制，需要新的编译和部署步骤以便于修改。让我们探索一种从外部源动态加载筛选器的方法。

1. 筛选器的拦截点

筛选器是在操作调用程序的内部解析每个操作方法的。有两个主要拦截点： GetFilters 和 InvokeActionMethodWithFilters 方法。这两个方法都被标记为受保护的和虚拟的。两个方法的签名如下：

```
protected virtual ActionExecutedContext InvokeActionMethodWithFilters(
    ControllerContext controllerContext,
    IList<IActionFilter> filters,
    ActionDescriptor actionDescriptor,
    IDictionary<string, object> parameters);

protected virtual FilterInfo GetFilters(
    ControllerContext controllerContext,
    ActionDescriptor actionDescriptor)
```

GetFilters 方法会早一些调用，并预期会返回用于指定操作的所有筛选器列表。在你的自定义调用程序中调用基方法 GetFilters 以后，便有了用于每个类别的可用筛选器的完整列表(一个包括异常、结果、授权和操作筛选器的列表)。注意 FilterInfo 类——System.Web.Mvc 中的一个公共类——提供了用于每个类别的特定筛选器集合：

```
public class FilterInfo
{
    // Private members
    ...

    public IList<IActionFilter> ActionFilters { get; }
    public IList<IAuthorizationFilter> AuthorizationFilters { get; }
    public IList<IExceptionFilter> ExceptionFilters { get; }
    public IList<IResultFilter> ResultFilters { get; }
}
```

InvokeActionMethodWithFilters 方法会在与操作方法性能有关的处理过程中被调用。在这种情况下，该方法只接收操作筛选器的列表(那些在用于方法的代码之前或之后执行的筛选器)。

要构建一种机制可以让开发人员更改那些应用于运行中方法的筛选器，我们的关注点应该集中在 InvokeActionMethodWithFilters 方法上。

2. 使用连贯代码添加操作筛选器

通过在自定义操作调用程序类中重写 InvokeActionMethodWithFilters 方法，就可以使用连贯代码来配置控制器和带有操作筛选器的控制器方法。下面的代码显示了如何在 Home 控制器的 Index 代码中动态添加 Compress 特性：

```
protected override ActionExecutedContext InvokeActionMethodWithFilters(
    ControllerContext controllerContext,
    IList<IActionFilter> filters,
    ActionDescriptor actionDescriptor,
    IDictionary<string, object> parameters)
{
  // Add the Compress action filter to the Index method of the Home controller
  if (actionDescriptor.ControllerDescriptor.ControllerName == "Home" &&
     actionDescriptor.ActionName == "Index")
  {
      // Configure the filter and add to the list
      var compressFilter = new CompressAttribute();
      filters.Add(compressFilter);
  }

  // Go with the usual behavior and execute the action
  return base.InvokeActionMethodWithFilters(
      controllerContext, filters, actionDescriptor, parameters);
}
```

可以从各个方面修改此代码。例如，可以支持区域并检查控制器的类型而非名称。此外，可以读取筛选器以便从配置文件中添加，也可以使用 IoC 容器来解析它们。通常来说，这种方法使你有机会动态配置筛选器，并且它还可以在控制器代码之外保持特性。

3. 自定义操作调用程序

让我们稍微概括一下这一概念。假设你创建了一个自定义操作调用程序，并重写了它的 InvokeActionMethodWithFilters 方法，如下所示：

```
protected override ActionExecutedContext InvokeActionMethodWithFilters(
    ControllerContext controllerContext,
    IList<IActionFilter> filters,
    ActionDescriptor actionDescriptor,
    IDictionary<String, Object> parameters)
{
   // Load (dynamic-loading) filters for this action
   var methodFilters = LoadFiltersForAction(actionDescriptor);
   if (methodFilters.Count == 0)
     return base.InvokeActionMethodWithFilters(controllerContext,
                                               filters, actionDescriptor, parameters);

   // Apply filter(s)
   foreach (var filter in methodFilters)
   {
      var filterInstance = GetFilterInstance(filter);
      if (filterInstance == null)
         continue;

      // Initialize filter (if params are specified)
      InitializeFilter(filter, filterInstance);

      // Add the filter
      filters.Add(filterInstance);
   }

   // Exit
   return base.InvokeActionMethodWithFilters(controllerContext,
                                             filters, actionDescriptor, parameters);
}
```

该代码包含了大量的占位符方法，实质上会读取可能在 web.config 文件中找到的动态定义的筛选器。提供的操作描述符用来找出有关当前操作的信息。操作详细信息用来读取定义筛选器相关的信息。最后，筛选器通过反射进行实例化和初始化，并添加到用于重写方法的筛选器集合。你要负责 web.config 节的结构。一段美观的 web.config 节可能会像下面这样：

```
<dynamicFilters>
  <action name="Home.Test">
    <filter type="MvcGallery3.Extensions.Filters.CompressAttribute,
        MvcGallery3.Extensions" />
    <filter type="MvcGallery3.Extensions.Filters.AddHeaderAttribute,
        MvcGallery3.Extensions">
```

```
      <param Name="Name" Value="X-MvcAspectFX" />
      <param Name="Value" Value="3222" />
    </filter>
  </action>
</dynamicFilters>
```

可以在本书的同步源代码中找到该动态加载筛选器框架的实现。

4. 注册自定义调用程序

每个控制器都有自己的通过公共属性公开的操作调用程序。可以在各个控制器的构造函数中替换操作调用程序。如果打算将自定义调用程序应用于任何控制器实例，可以考虑将代码移到控制器工厂。我常常采用一个如下面示例中所定义的自定义控制器基类，从那里获得需要动态加载筛选器的任何控制器：

```
public class AxpectController : Controller
{
    public AxpectController()
    {
        ActionInvoker = new AxpectActionInvoker();
    }
}

public class HomeController : AxpectController
{
    // Attributes for this method come from global filters defined in global.asax
    // and dynamic-loading filters defined in web.config.
    public ActionResult Index()
    {
       return View();
    }
}
```

使用一个用于控制器的自定义基类不是所有开发人员都喜欢的一种编程技术。这可能有几个原因，包括个人偏好，但通常是因为单继承语言(比如微软 C#和微软 Visual Basic .NET)仅提供一个继承点，它往往会明智地将其保留供以后使用。

如果不喜欢使用控制器基类来启用动态加载的筛选器，则可以在筛选器提供程序之上构建一个替代项。

5. 通过筛选器提供程序启用动态加载

OnActionExecuting 方法是在操作方法运行之前执行某些在先前已注册的操作筛选器中定义的自定义代码的正确地方。不过，在这个点添加新的筛选器执行某些自定义代码已经太

晚了。除了使用自定义操作调用程序，筛选器提供程序也是一个系统定义的挂钩，让你可以基于运行时条件动态注册筛选器。

筛选器提供程序是一个实现下列接口的类：

```
public interface IFilterProvider
{
    IEnumerable<Filter> GetFilters(
    ControllerContext controllerContext, ActionDescriptor actionDescriptor);
}
```

GetFilters 方法被操作调用程序调用，为指定操作动态地加载筛选器。因此自定义筛选器提供程序正是实现不再需要显式调整操作调用程序和控制器类的工具。需要在 global.asax 中注册你自己的筛选器提供程序，如下所示：

```
protected void Application_Start()
{
    ...
    RegisterFilterProviders(FilterProviders.Providers);
}
public static void RegisterFilterProviders(FilterProviderCollection
    providers)
{
    providers.Add(new DynamicLoadingFilterProvider());
}
```

这是就配置而言需要准备到位的所有代码。接下来，需要在配置文件中添加信息(如前面的示例代码所完成的)，并且运行应用程序。下面是用于筛选器提供程序的代码：

```
public class DynamicLoadingFilterProvider : IFilterProvider
{
    public IEnumerable<Filter> GetFilters(
      ControllerContext controllerContext, ActionDescriptor actionDescriptor)
    {
      // The method reads from web.config and returns a collection of Filter objects
        return LoadFiltersFromConfiguration(actionDescriptor);
    }
}

public static IList<Filter> LoadFiltersFromConfiguration(ActionDescriptor
    actionDescriptor)
{
    var methodFilters = LoadFiltersForAction(actionDescriptor);
```

```
        var filters = new List<Filter>();
        foreach (var filterName in methodFilters)
        {
            // Instantiate and initialize the filter (if params are specified)
            var filterInstance = GetFilterInstance(filterName);
            if (filterInstance == null)
                continue;
           InitializeFilter(filterName, filterInstance);

            // Add to the list to return
            var filter = new Filter(filterInstance, FilterScope.Action, -1);
            filters.Add(filter);
        }

        return filters;
    }
```

筛选器提供程序通过一个封装类——Filter 类——来管理筛选器。该类封装了实际的操作筛选器实例，外加一个用于顺序(在本例中为–1)和筛选器范围的值。这个范围在 FilterScope 枚举类型中定义。该类型是一个系统类型，定义如下：

```
    public enum FilterScope
    {
        First = 0,        // First to run
        Global = 10,      // Applies to all controllers/actions
        Controller = 20,  // Applies at controller level
        Action = 30,      // Applies at action level
        Last = 100        // Last to run
    }
```

有了筛选器提供程序、下面的代码、再加上之前显示的 web.config 内容，调用 Test 方法时就会产生如图 8-4 所示的输出：

```
    // Dynamically bound to action filters via web.config
    // (Adding a custom header.)
    public ActionResult Test()
    {
        return View();
    }
```

在此示例中，我们通过使用 global.asax 中的代码来注册筛选器提供程序。或者，可以通过使用自己的依赖性解析器来注册筛选器提供程序。

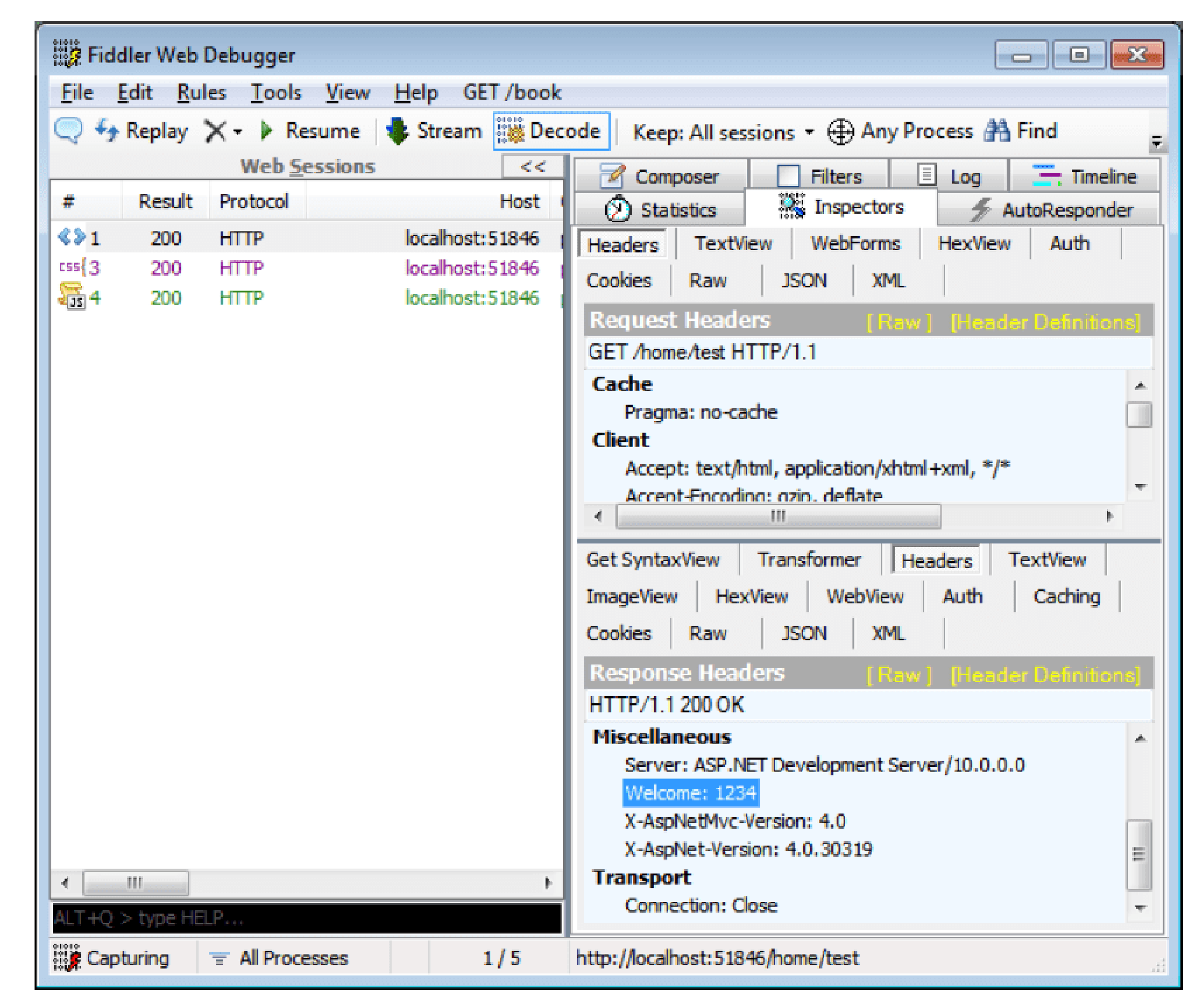


图 8-4　通过动态添加筛选器来修改请求的响应

8.3　操作结果类型

迄今为止你在这本书里所看到的大部分示例中，控制器方法总是返回一个 ActionResult 对象。这其实是 ASP.NET MVC 提供的代表操作结果的基类。根据用于返回操作结果的方法(比如 View、PartialView 方法等)，实际的类型会采用不同的形式。例如，当一个方法返回 HTML 标记时，实际的操作结果类型就是 ViewResult。

操作结果类型也是在很大程度上可以自定义的 ASP.NET MVC 的一个特性。所以，让我们继续进行到下一阶段，考虑可以处理的用来自定义操作结果的工具。

8.3.1　内置的操作结果类型

当由控制器操作方法返回的 ActionResult 对象进一步由操作调用程序处理时，会生成浏览器的响应并将其写入输出流。换句话说，ActionResult 类型，以及直接从它派生而来的任何类型，并不代表被发送到浏览器的真正响应。它只是一个包含特定响应数据(比如标头、cookies、状态代码、内容类型，以及任何可用于生成响应的数据)的封装类，并且它知道如何处理这些数据以生成浏览器的实际响应。

ActionResult 对象定义如下：

```
public abstract class ActionResult
{
    protected ActionResult()
    {
    }

    public abstract void ExecuteResult(ControllerContext context);
}
```

ExecuteResult 方法提供了向浏览器呈现结果的逻辑。为了便于理解操作结果对象的机制，查看几个内置于 ASP.NET MVC 中的操作结果类是很有用的。

1. 返回自定义状态代码

一个最简单的操作结果类是 HttpStatusCodeResult 类。此类代表了一个设置了特定 HTTP 状态代码和描述的操作响应：

```
public class HttpStatusCodeResult : ActionResult
{
    public HttpStatusCodeResult(Int32 statusCode) : this(statusCode, null)
    {
    }
    public HttpStatusCodeResult(Int32 statusCode, String statusDescription)
    {
        StatusCode = statusCode;
        StatusDescription = statusDescription;
    }
    public override void ExecuteResult(ControllerContext context)
    {
        if (context == null)
            throw new ArgumentNullException("context");

        // Prepare the response for the browser
        context.HttpContext.Response.StatusCode = StatusCode;
        if (StatusDescription != null)
            context.HttpContext.Response.StatusDescription = StatusDescription;
    }
}
```

可以看到，它所做的只是设置响应的状态代码和对响应的描述。下面显示了如何使用此类来返回一个 HTTP 403 代码(Forbidden)，如图 8-5 所示，意思是说服务器理解此请求，但有充足的理由拒绝执行它：

```
public ActionResult Forbidden()
{
    return new HttpStatusCodeResult(403);
}
```

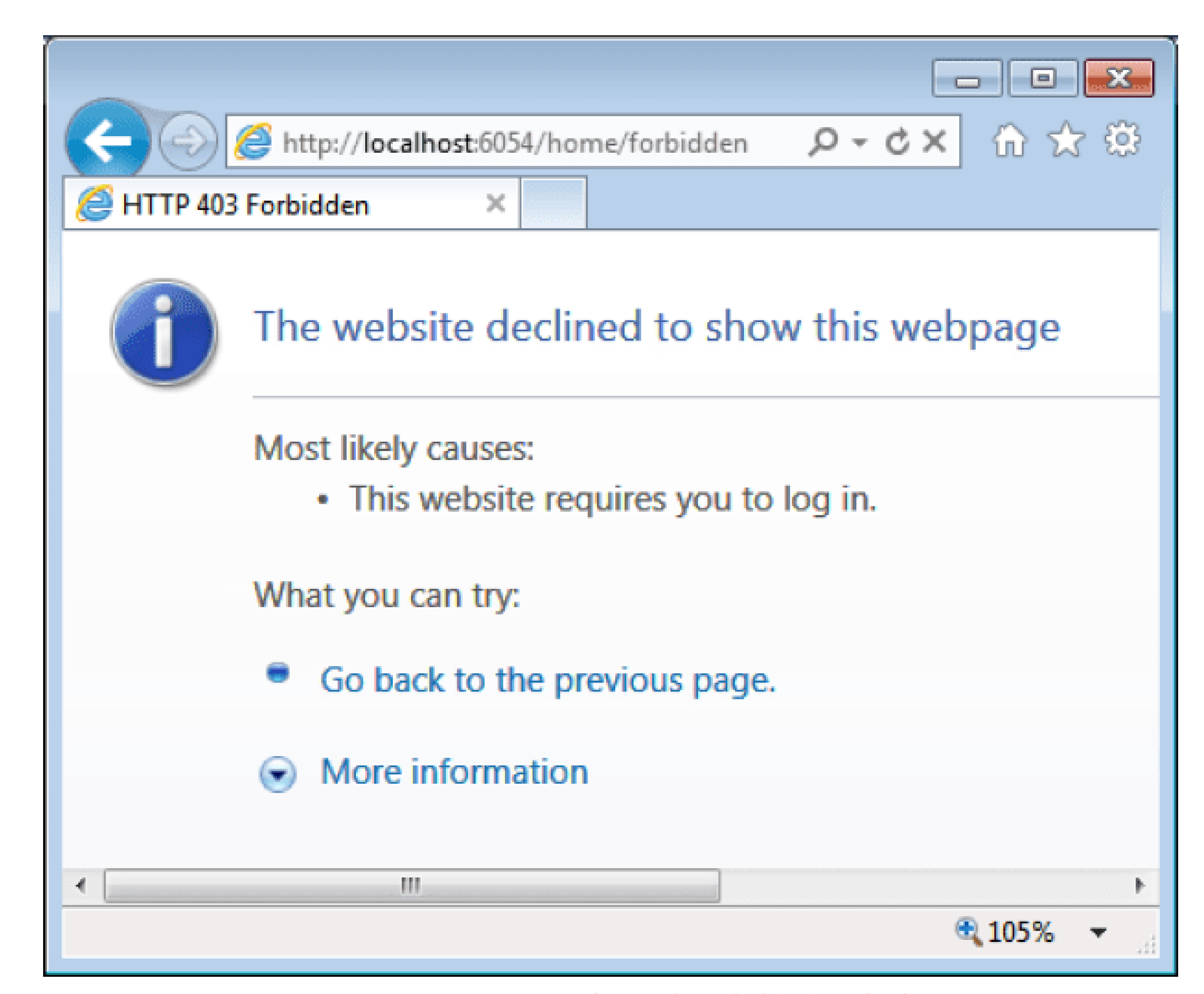


图 8-5　Web 服务器拒绝执行请求

当你需要从应用程序中的几个地方返回相同的状态代码时，最好将状态代码封装在一个自定义的操作结果类中。它并不会带来多大的改变，却一定会增加可读性。ASP.NET MVC 在 401 代码(未经授权)中也遵循这种模式，并且为你提供了 HttpUnauthorizedResult 类，它是由 Authorize 操作筛选器使用的其中一个类。下面是另一个同样流行的 404 代码的示例：

```
public class HttpNotFoundResult : HttpStatusCodeResult
{
    public HttpNotFoundResult() : this(null)
    {}
    public HttpNotFoundResult(String statusDescription) : base(404,
        statusDescription)
    {}
}
```

即使整体上简化了，这些类也显示出了操作结果类的核心结构。

2. 返回 JavaScript 代码

一个稍微复杂点的例子是 JavaScriptResult 类。此类提供了一个公共属性——Script 属性——它包含要写入输出流的脚本代码(类似于字符串)，如下所示：

```
public class JavaScriptResult : ActionResult
{
    public String Script { get; set; }

    public override void ExecuteResult(ControllerContext context)
    {
        if (context == null)
            throw new ArgumentNullException("context");

        // Prepare the response
        var response = context.HttpContext.Response;
        response.ContentType = "application/x-javascript";
        if (Script != null)
            response.Write(Script);
    }
}
```

从如下所示的操作方法中使用 JavaScriptResult 类：

```
public ActionResult Javascript()
{
   var script = "function helloWorld() {alert('hello, world!');}";
   var result = new JavaScriptResult {Script = script};
   return result;
}
```

对于一些预定义的操作结果，ASP.NET MVC 提供了帮助器方法，它们隐藏了操作结果对象的创建过程。可以以更简洁的方式重写前面的代码，如下所示：

```
public ActionResult Javascript()
{
   var script = "function helloWorld() {alert('hello, world!');}";
   return JavaScript(script);
}
```

当用户调用一个返回 JavaScript 的操作时，就会显示如图 8-6 所示的经典下载面板。

事实证明操作方法返回 JavaScript 的正确位置是<script>标记。

```
<script type="text/javascript" src="/home/javascript"></script>
```

加上一行如在刚才的 HTML 中所示的代码以后，就可以以编程方式调用正在下载的任何脚本函数了。

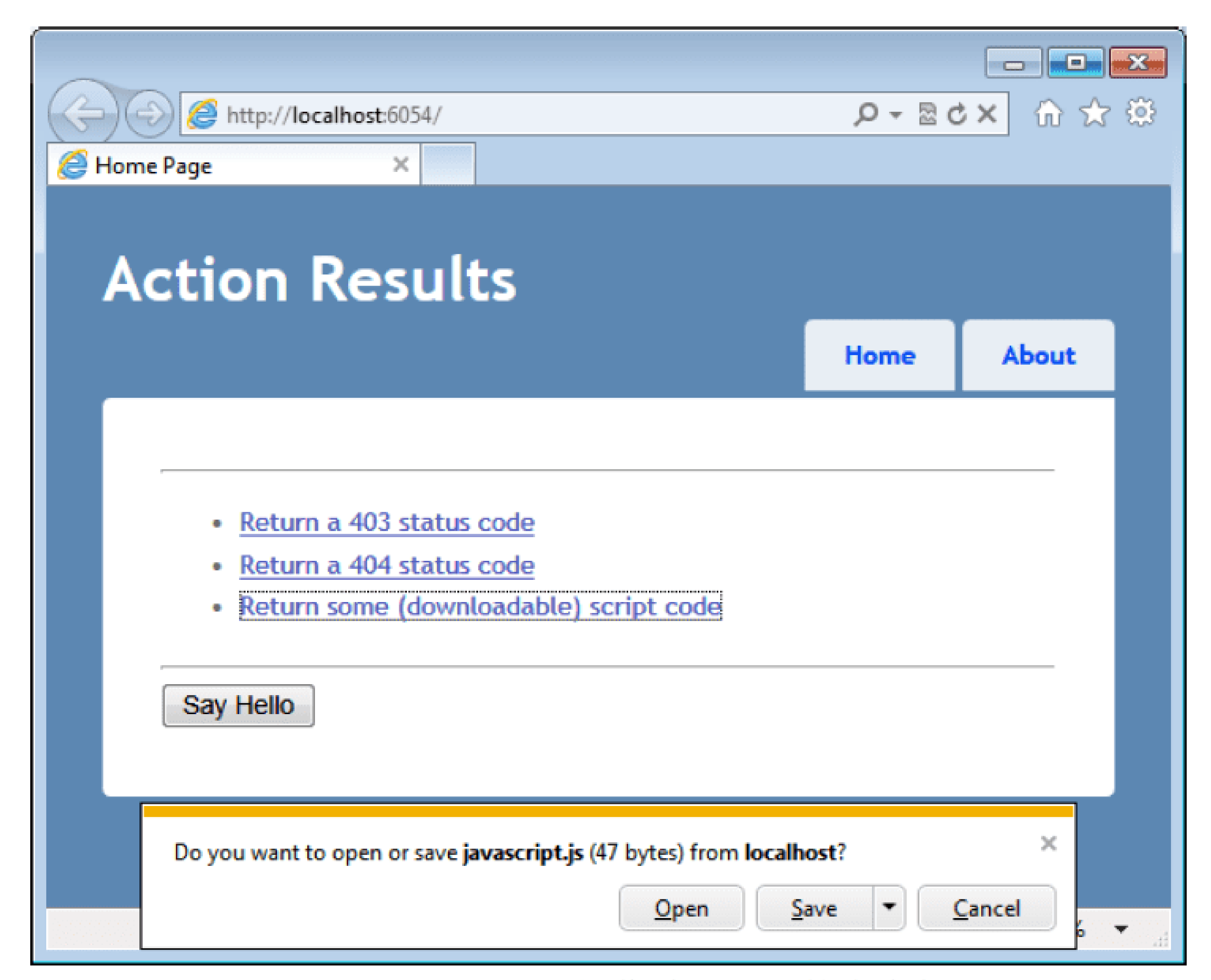


图 8-6　由 JavaScript 操作结果返回的脚本代码

3. 返回 JavaScript 对象标记数据

要从 ASP.NET MVC 控制器类内部返回 JavaScript 对象标记(JSON)数据，你只需要一个返回 JsonResult 对象的操作方法。JsonResult 类会获取任意.NET 对象，并尝试通过使用 JavaScriptSerializer 系统类将其序列化为 JSON。下面是对服务 JSON 数据的操作方法的一个可能定义：

```
public JsonResult GetCountries(String area)
{
    // Grab some data to serialize and return
    var countries = CountryRepository.GetAll(area);
    return Json(countries);
}
```

Json 方法是在 Controller 类上定义的，它会在内部创建一个 JsonResult 对象。JsonResult 对象的目的是要将指定的.NET 对象——在本示例中是一份国家名单——序列化为 JSON 格式。Json 方法有几个重载，通过它们可以指定所需的内容类型字符串(默认为 application/json)，并请求行为。请求行为包括通过 HTTP GET 请求允许或拒绝 JSON 内容。

默认情况下，ASP.NET MVC 不通过 HTTP GET 请求来提供 JSON 内容。这意味着如果前面的代码是在一个 GET 上下文中调用的，则会执行失败，而这是进行 JSON 调用的最明显的方式。如果认为你的方法是安全的，不会有潜在的泄露敏感数据的风险，你就可以修改代

码，如下所示：

```
public JsonResult GetCountries()
{
    // Grab some data to serialize and return
    var countries = CountryRepository.GetAll();
    return Json(countries, JsonRequestBehavior.AllowGet);
}
```

就像脚本一样，如果操作方法通过浏览器以交互方式调用，则从该方法返回的 JSON 数据就会被下载。如果将操作方法嵌入到<script>标记或通过 Ajax 调用，内容就可被用作原始数据。下面的脚本显示了通过 ASP.NET MVC 控制器下载 JSON 数据的一个 Ajax 调用。下载的数据随后会被用来填充一个名为 listOfCountries 的下拉列表。

```
function downloadCountries(area) {
    $("#listOfCountries").empty();

    $.getJSON("/home/getcountries", { area: "EMEA" },
            function (data) { _displayCountries(data); });
}

function _displayCountries(data) {
    // Get the reference to the drop-down list
    var listbox = $("#listOfCountries")[0];

    // Fill the list
    for (var i = 0; i < data.length; i++) {
        var country = data[i];
        var option = new Option(country.Name, country.Code);
        listbox.add(option);
    };
}
```

该视图中的 HTML 看起来如图 8-7 所示。

4. 返回基本类型

严格地讲，控制器操作方法不会被强制返回一个 ActionResult 对象。例如，可以让它返回一个字符串或者一个整数，以及在方法签名中作为声明的返回类型的 String 与 Int32。然而，请注意，无论你返回什么类型，都会被 ASP.NET MVC 框架封装在一个 ContentResult 对象中。

相反，如果是虚方法，操作结果就会是一个 EmptyResult 对象。通过使用操作筛选器，可以随意修改结果对象及它的参数。所以最终，你仍然可以声明控制器方法不返回任何内容，而是定制成返回一个附加有操作筛选器的以编程方式返回一个指定结果对象的值。

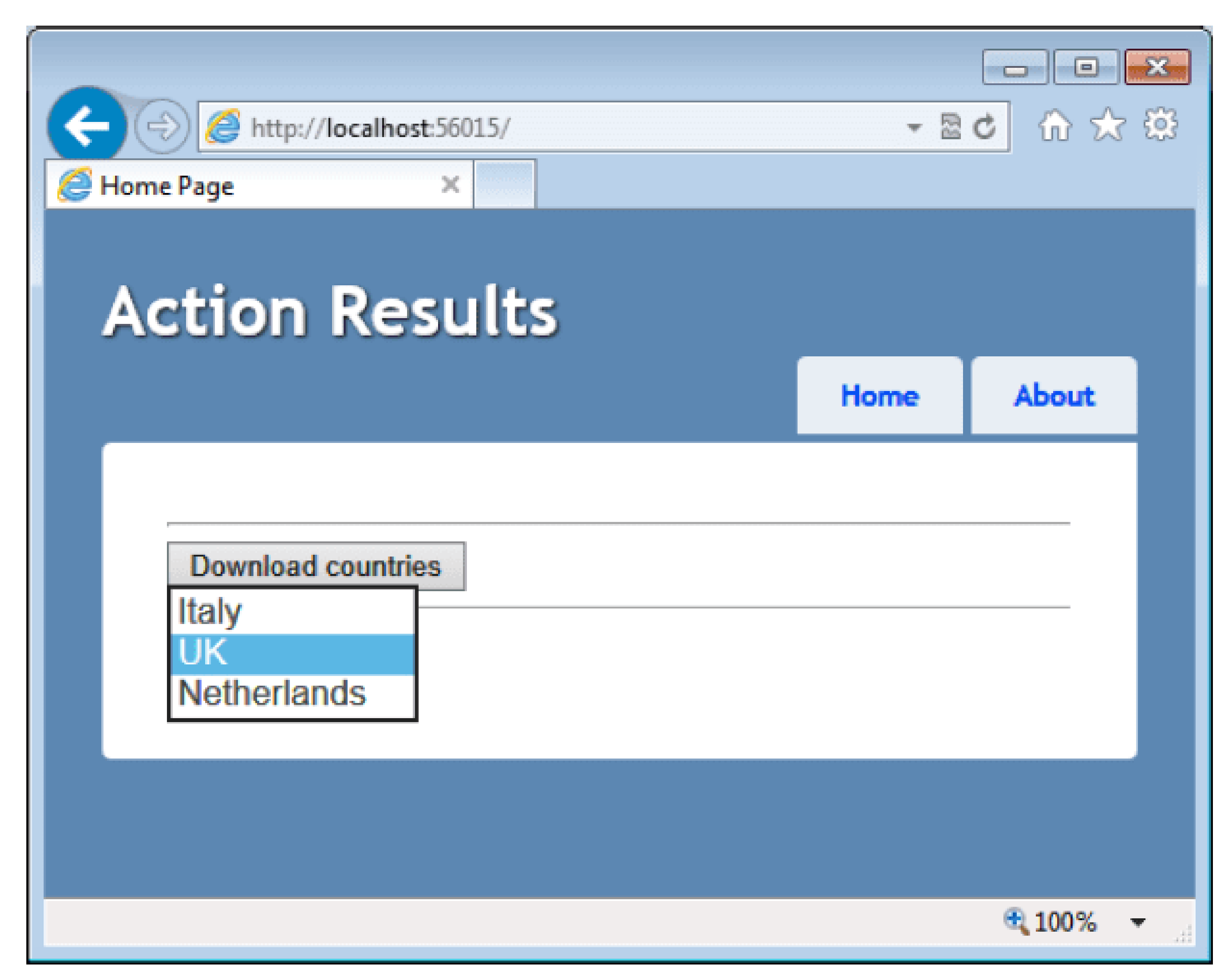


图 8-7　由 Ajax 检索的 JSON 数据填充的下拉列表

8.3.2　自定义结果类型

归根结底，操作结果对象是封装需要在特殊情况下完成的所有任务的一种方式，比如当请求的资源丢失或被重定向时，或者当某些特殊的响应必须提供给浏览器时。让我们检查几个具有自定义操作结果对象的相关方案。

1. 返回 JSONP 字符串

你可能知道，浏览器通常不允许页面放置会调用另一个域的网站的 Ajax。此规则是相对较新的，添加它用来将恶意用户的表面攻击区域减少到零。这种限制的实际效果是，阻止对网站(承载在不同的域而非请求页面)进行 Ajax 调用并下载 JSON 数据。更有意思的是，这一限制单方面被浏览器采用，却忽略了远程站点能否提供数据。

这对 Ajax 调用和 JSON 数据来说还真是麻烦，但我们仍然不能控制可以从任何可访问网站随意下载的图片和脚本文件。正是浏览器的这一特点可以被利用来通过 Ajax 从显式允许的站点安全地下载 JSON 数据。这就是 JSONP(带填充的 JSON)协议的核心内容。

JSONP 是某些网站用来公开其 JSON 内容的一项协议，这是以一种让调用方更易于通过脚本甚至从外部域使用数据的方式来公开的。其原理是返回封装在一个脚本函数调用中的 JSON 内容。换句话说，网站会返回下面的字符串，而非包含普通 JSON 数据的字符串：

```
yourFunction("{...}"); // instead of plain {...} JSON data
```

从<script>标记中调用网站 URL 后，浏览器会收到一个带有固定输入字符串的看起来像普通函数的调用。这时，输入数据是文本还是 JSON 文本对浏览器来说已经没有关系了。

支持 JSONP 的网站会公开一个用于调用的公共方式，指明在返回 JSON 数据时要使用的 JavaScript 函数的封装名称。无须多说，JavaScript 函数对发出请求的站点来说必须是本地的，并且应该要包含处理 JSON 数据的逻辑。让我们看看如何在 ASP.NET MVC 控制器中添加 JSONP 功能。请思考下面的示例：

```
public ActionResult GetCountriesJsonp(String area)
{
    var result = new JsonpResult
                      {
                         JsonRequestBehavior = JsonRequestBehavior.AllowGet,
                         Data = GetCountries(area)
                      };
    return result;
}
```

JsonpResult 类扩展了本地 JsonResult，并将正在返回的 JSON 字符串封装成一个对指定 JavaScript 函数的调用。

```
public class JsonpResult : JsonResult
{
    // The callback name here is the parameter name to be added to the URL to
specify the name of the JavaScript function padding the JSON response. This name
is arbitrary and is part of your site's SDK.
    private const String JsonpCallbackName = "callback";

    public override void ExecuteResult(ControllerContext context)
    {
        if (context == null)
            throw new ArgumentNullException("context");

        if ((JsonRequestBehavior == JsonRequestBehavior.DenyGet) &&
             String.Equals(context.HttpContext.Request.HttpMethod, "GET"))
            throw new InvalidOperationException();

        var response = context.HttpContext.Response;
        if (!String.IsNullOrEmpty(ContentType))
            response.ContentType = ContentType;
        else
            response.ContentType = "application/json";
        if (ContentEncoding != null)
            response.ContentEncoding = this.ContentEncoding;

        if (Data != null)
        {
```

```
            String buffer;
            var request = context.HttpContext.Request;
            var serializer = new JavaScriptSerializer();
            if (request[JsonpCallbackName] != null)
               buffer = String.Format("{0}({1})", request[JsonpCallbackName],
                                                    serializer.Serialize(Data));
            else
               buffer = serializer.Serialize(Data);

            response.Write(buffer);
        }
    }
}
```

这个类与 JsonResult 几乎是相同的，除了在 ExecuteResult 方法中有一点小变化以外。在序列化为 JavaScript 之前，该代码会检查常规的 JSONP 参数是否已经通过请求并相应地确定了 JSON 字符串。图 8-8 显示了以下 URL 的响应正文：

```
/home/getcountriesjsonp?callback=_cacheForLater&area=asia
```

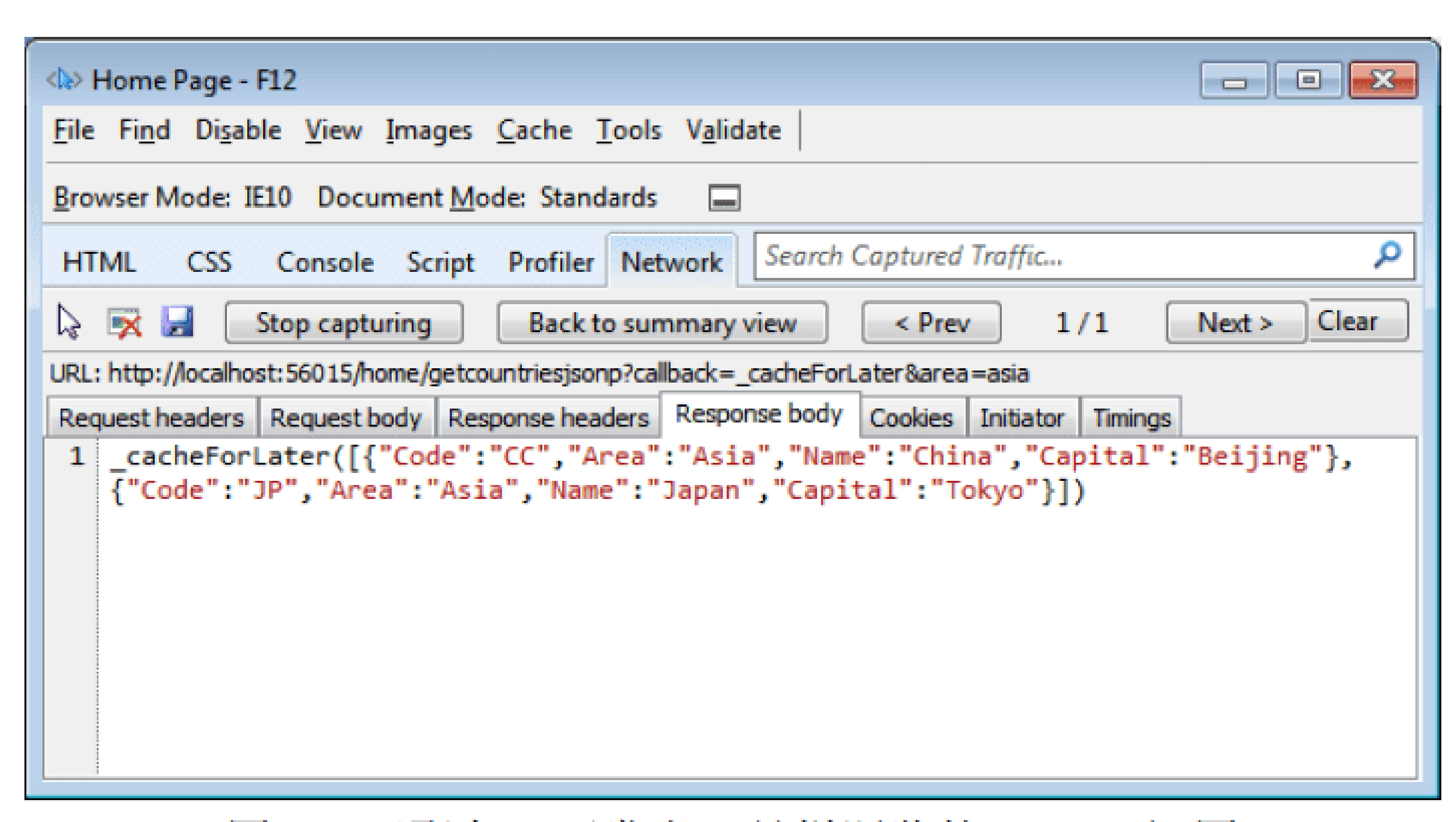


图 8-8　通过 IE 开发者工具栏浏览的 JSONP 视图

要放置一个 JSONP 调用，可以遵循几条路由。例如，可以手动设置一个<script>标记，使 JSON 数据对页面全局可用：

```
<script type="text/javascript"
        src="/home/getcountriesjsonp?callback=_cacheForLater&area=asia">
    </script>
```

或者，可以通过 jQuery 或普通的 JavaScript 发起一个 Ajax 调用。尤其是，jQuery 库可以通过 getJSON 函数提供具体的支持。当你传递给 getJSON 的 URL 包含一个 xxx =?段时，jQuery 就会将其解释为 JSONP 调用并以一种特殊方式处理它。更确切地说，jQuery 会动态创建一个

<script>标记，并通过它下载响应。填充函数的名称会通过 jQuery 动态生成，并且会被指向你提供的 getJSON 回调代码。下面是一个示例：

```
$.getJSON("/home/getcountriesjsonp?callback=?", { area: "NA" },
            function (data) { _displayCountries(data); });
```

URL 中的回调名称必须与你调用的网站所定义的公共 JSONP 名称相匹配。我在这个示例中使用回调是因为那是由先前提供的 JsonpResult 实现所识别的名称。

2. 返回联合源

如果在网络中搜索一个不寻常的操作结果的例子，很可能会在搜索列表的顶部找到一个联合操作结果对象。让我们简单地看看这个流行的例子。

SyndicationResult 类支持 RSS 2.0 和 ATOM 1.0，它为你提供了一个方便的属性可以以编程方式选择其中的一个。默认情况下，该类会生成 RSS 2.0 源。要编译此示例，你需要引用 System.ServiceModel 程序集并导入 System.ServiceModel.Syndication 名称空间：

```
public class SyndicationResult : ActionResult
{
    public SyndicationFeed Feed { get; set; }
    public FeedType Type { get; set; }

    public SyndicationResult()
    {
        Type = FeedType.Rss;
    }
    public SyndicationResult(
        string title, string description, Uri uri, IEnumerable
                                          <SyndicationItem> items)
    {
        Type = FeedType.Rss;
        Feed = new SyndicationFeed(title, description, uri, items);
    }
    public SyndicationResult(SyndicationFeed feed)
    {
        Type = FeedType.Rss;
        Feed = feed;
    }

    public override void ExecuteResult(ControllerContext context)
    {
        // Set the content type
        context.HttpContext.Response.ContentType = GetContentType();
```

```
            // Create the feed, and write it to the output stream
            var feedFormatter = GetFeedFormatter();
            var writer = XmlWriter.Create(context.HttpContext.Response.Output);
            if (writer == null)
               return;
            feedFormatter.WriteTo(writer);
            writer.Close();
        }

        private String GetContentType()
        {
            if(Type == FeedType.Atom)
               return "application/atom+xml";
            return "application/rss+xml";
        }

        private SyndicationFeedFormatter GetFeedFormatter()
        {
            if (Type == FeedType.Atom)
               return new Atom10FeedFormatter(Feed);
            return new Rss20FeedFormatter(Feed);
        }
    }

    public enum FeedType
    {
        Rss = 0,
        Atom = 1
    }
```

该类会得到一个联合源，并通过使用 RSS 2.0 或 ATOM1.0 格式将其序列化到客户端。创建一个可使用的源就是另一回事了，但它也是一个涉及控制器而不是基础架构的问题。下面显示了如何编写一个返回源的控制器方法：

```
    public SyndicationResult Feed()
    {
        var items = new List<SyndicationItem>();
        items.Add(new SyndicationItem(
            "Controller descriptors",
            "This post shows how to customize controller descriptors",
            null));
        items.Add(new SyndicationItem(
            "Action filters",
```

```
            "Using a fluent API to define action filters",
            null));
        items.Add(new SyndicationItem(
            "Custom action results",
            "Create a custom action result for syndication data",
            null));
        var result = new SyndicationResult(
            "Programming ASP.NET MVC",
            "Dino's latest book",
             Request.Url,
             items);

        result.Type = FeedType.Atom;
        return result;
    }
```

你要创建一个 SyndicationItem 对象的列表，并为其中的每一个提供标题、内容和一个备用链接(在代码片段中为 null)。通常要从应用程序中可能有的一些存储库中检索这些项。最后，将这些项传递给 SyndicationResult 对象，顺带着标题和对将要创建并序列化的源的描述。图 8-9 显示了一个 IE 中的 ATOM 源。

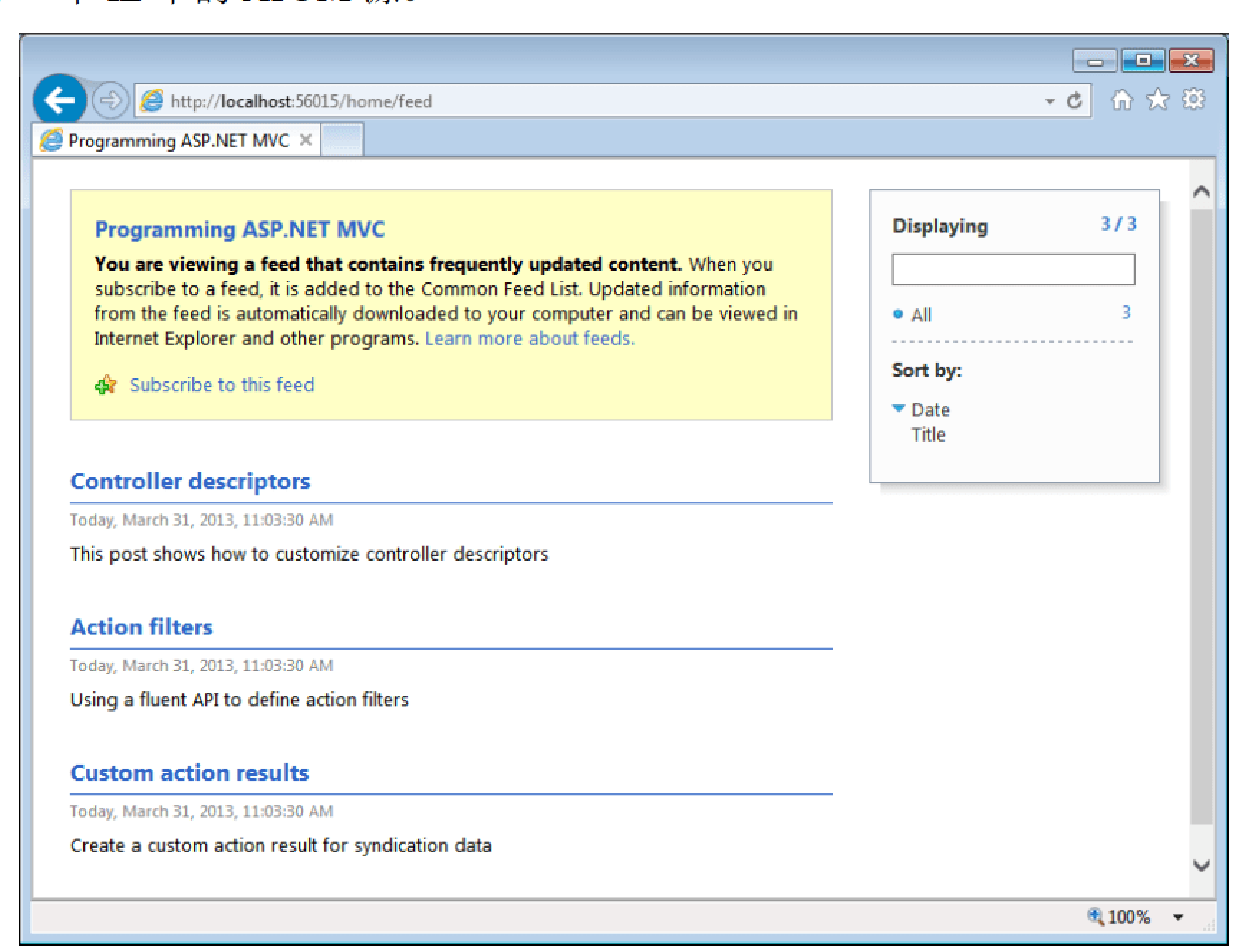


图 8-9　显示在 IE 中的 ATOM 源

3. 处理二进制内容

开发人员的常见需求是从请求返回二进制数据。许多不同类型的数据均是以二进制数据来表示的，比如图片的像素、PDF 文件的内容，甚至是微软的 Silverlight 程序包。

你并不真的需要一个特设的操作结果对象来处理二进制数据。在内置的操作结果对象中，你肯定可以找到在处理二进制数据时能够帮助到你的那个对象。如果要转换的内容存储在磁盘文件中，则可以使用 FilePathResult 对象。如果内容可以通过数据流获取，就使用 FileStreamResult 并选择 FileContentResult，如果可以让它用作字节数组的话。

所有这些对象都从 FileResult 派生，只是在如何将数据写出到响应流方面相互不同。让我们回顾一下 ExecuteResult 在 FileResult 内是如何实现的。

```
public override void ExecuteResult(ControllerContext context)
{
    if (context == null)
        throw new ArgumentNullException("context");

    var response = context.HttpContext.Response;
    response.ContentType = this.ContentType;
    if (!String.IsNullOrEmpty(this.FileDownloadName))
    {
        var headerValue = ContentDispositionUtil.
                                        GetHeaderValue(FileDownloadName);
        context.HttpContext.Response.AddHeader("Content-Disposition",
                                        headerValue);
    }

    // Write content to the output stream
    WriteFile(response);
}
```

该类有一个名为 ContentType 的公共属性，通过它可以与 MIME 类型的响应通信，这种响应的所有工作都是通过一种抽象方法——WriteFile——来进行的，从 WriteFile 派生的类都必须重写。

基类 FileResult 还通过 Content-Disposition 标头支持客户端浏览器中的 Save As 对话框。FileDownloadName 属性会指定文件将在浏览器中的 Save As 对话框中给出的默认名称。Content-Disposition 标头具有如下格式，其中 XXX 代表 FileDownloadName 属性的值：

```
Content-Disposition: attachment; filename=XXX
```

注意文件的名称应该在 US-ASCII 字符集中，允许没有目录路径信息。最后，MIME 类型一定是对浏览器未知的；否则，将使用注册的处理程序来处理内容。

基类 FileResult 与派生的类之间的差异主要涉及 WriteFile 方法的实现。尤其是，FileContentResult 会将字节数组直接写入输出流，如下所示：

```
// FileContents is a property on FileContentResult that points to the bytes
protected override void WriteFile(HttpResponseBase response)
{
   response.OutputStream.Write(FileContents, 0, FileContents.Length);
}
```

FileStreamResult 提供了一个不同的实现。它有一个 FileStream 属性，提供了数据的读取，且 WriteFile 中的代码会以缓冲的方式读写。

```
protected override void WriteFile(HttpResponseBase response)
{
    Stream outputStream = response.OutputStream;
    using (FileStream)
    {
        byte[] buffer = new byte[0x1000];
        while (true)
        {
            var count = FileStream.Read(buffer, 0, 0x1000);
            if (count == 0)
                return;
            outputStream.Write(buffer, 0, count);
        }
    }
}
```

最后，FilePathResult 会将现有文件复制到输出流。WriteFile 的实现语句在这种情况下是相当短的。

```
// FileName is the name of the file to read and transmit
protected override void WriteFile(HttpResponseBase response)
{
    response.TransmitFile(FileName);
}
```

有了这些可用的类，你就可以处理任何形式的需要从 URL 以编程方式提供的二进制数据。下面的代码显示了如何提供一个图片：

```
public ActionResult Img()
{
    const String file = "stones.jpg";
    return File(Server.MapPath(String.Format("~/content/images/{0}", file)),
        "image/jpeg");
```

```
}
```

4. 返回 PDF 文件

作为最后一个例子，我们看看怎样才能返回一些 PDF 数据。老实说，到这一步，要解决的最大问题是你如何获取 PDF 内容。如果 PDF 内容是静态资源，比如服务器文件，那么只需要使用 FilePathResult 即可。更好的是，可以使用特设的 File 方法，如下所示：

```
public ActionResult About()
{
    ...
    return File(fileName, "application/pdf");
}
```

要动态创建 PDF 内容，可以使用许多库，比如 iTextSharp (参见 http://sourceforge.net/projects/itextsharp)。一些商业产品以及各种开源项目也可以从 HTML 内容创建 PDF 内容。这个选项对于 ASP.NET MVC 应用程序来讲尤其有用，因为可以攫取一个局部视图，将其变成可下载的 PDF 内容。

更常见的情形是，你需要创建并返回从模板改编的 PDF 文档。有两种主要的方式可以考虑：使用 Office 自动化并且从微软 Word 或 Excel 文档创建 PDF；或者使用 Reporting Services。两相比较，使用微软 Office 也许更易于进行；Reporting Services 则更可靠，并且可以免除隐藏的成本和不易察觉的问题。顺带说一下，微软本身不鼓励在服务器应用程序中使用 Office 自动化(参见 http://support.microsoft.com/?id=257757)。我已经将其用于几个应用程序且并没有遭遇任何严重的问题。但是，我的经验可能是个例而非通用法则。

本书的同步代码提供了一个示例的 ASP.NET MVC 项目，当你调用一个指定操作方法时，它就会提供一个 PDF 文件。该代码使用了 Office 自动化，从 DOTX 模板创建新的 Word 文档。该模板包含了一些以编程方式替换为用户定义值的书签。图 8-10 显示了由前面的代码所创建的示例 PDF 文档。

注意：

示例 PDF 应用程序是通过在 Web 服务器磁盘上创建本地文件来工作的。只要你是通过使用微软 Visual Studio 环境(以及嵌入式的 Visual Studio Web 服务器)来测试应用程序，一切都会很顺利。当你是在真正的 IIS Web 服务器下开始测试时，由于默认的安全权限，本地保存文件可能会引发一些问题。ASP.NET 默认账户并不会在创建该文件的文件夹上具有写入权限。要解决此问题，需要提高 ASP.NET 账户在你打算创建临时或持久的 PDF 文件的文件夹中的写入权限。

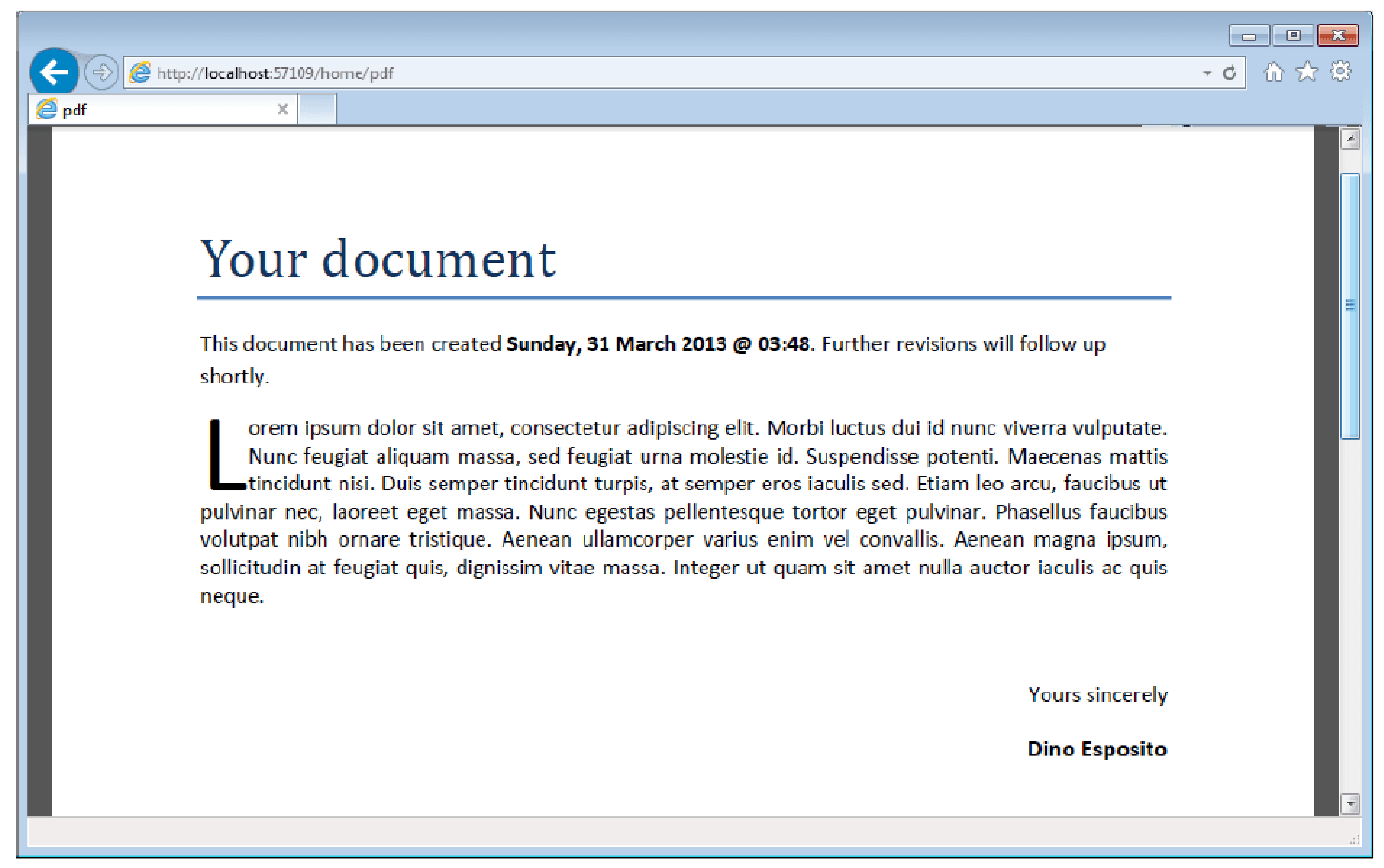


图 8-10　动态创建 PDF 文档的示例应用程序

8.4　本章小结

不论你是否喜欢 ASP.NET MVC，都不能否认它是一个高度自定义和可扩展的框架。在 ASP.NET MVC 中，可以充分控制每个操作的执行，并且在请求被处理之前以及被处理之后进行干预。同样，对于发出对客户端浏览器的响应的过程，你几乎可以获得所有方面的控制权。

即使可以自定义 ASP.NET MVC 的每一个方面，也不会总想对它们进行重写。这一章中所讨论的自定义方面是那些我认为常被自定义的方面，以及更重要的是，那些能带来巨大收益的方面，如果能够恰当地自定义的话。

第 9 章

ASP.NET MVC 中的测试与可测试性

在准备作战时我总是发现计划一无是处，但计划却是必不可少的。

——*Dwight D. Eisenhower*

在.NET 刚推出时，应用程序的平均复杂性并不是很高，微软 Visual Studio 调试工具就足以用来达成测试目的了。快速应用程序开发(Rapid Application Development，RAD)是大多数人选择的范式；随之而来的就是，很少有开发人员会去关心编写测试程序了。

.NET 平台的成功使整个行业范围的许多企业都需要获取新的业务应用程序。这样一来，他们就抛弃了大量的基于不同开发团队的各种复杂性和业务规则。在只有 RAD 范式支持的情况下，生产力的提高变得越来越难。这最终导致了开发处理顺序的完全变更。开发人员除了要关注上市时间，还必须更多关注可维护性和可扩展性。当然，可维护性也带来了对代码可读性的要求，以处理日益增长的需求变化。

测试软件的能力，尤其是自动测试软件的能力，是非常重要的一个方面，因为自动化测试给了你一种机械方式，可以快速可靠地了解在某一时点运行的某些功能在做了一些更改之后是否仍然在正常运行。此外，测试还使你有可能计算出指标并时时把握项目的脉搏。归根结底，它带来的巨大变化是我们不用再在不可能成功的软件项目上浪费金钱了。测试是带来这一变化的重要一点。

无可否认生产力非常重要，但仅仅把重点放在生产力上成本太高，因为它会导致难以维护的低质量代码，维护费用也相当昂贵。所以如果是这样的话，又有什么好处呢？

用自动化方式测试软件的必要性——我们可以称之为采用 RAD 范式进行测试的必要性——提出了另一个关键点：拥有易于测试的软件的必要性。在这一章，我首先会明确一款软件具有可测试性所需的技术特点。接下来，介绍单元测试的基本知识——固件、断言、测试替身和代码覆盖率——最后以一些特定于 ASP.NET MVC 的单元测试的例子结束。

注意：

很长时间以来，许多.NET 开发人员都把“测试”和“调试”这样的术语看作是同义词。

调试的操作涉及捕捉和修复应用程序中的错误和异常。调试由一名开发人员实施，且大多是交互的多阶段过程。而测试的操作涉及确保代码某些部分的表现如预期那样运行。测试由特设程序实施，本质上是无人值守的任务，它是自动执行且集成在软件的构建过程中的。这两个进程是正交的，相互之间并不排斥。测试会是重现 bug 的有效办法；另一方面，它又能很好地设法预防 bug 的出现。

9.1 可测试性和设计

在软件架构的上下文中，对于可测试性一个广泛认可的定义是“执行测试的容易程度”。测试，当然是对软件的检查过程，确保软件表现一如预期、不包含错误、并且满足需求。

测试软件在概念上很简单：强制程序作用于正确的、错误的、丢失的或不完整的数据，并验证结果是否符合预期。那么如何强制程序作用于你的输入数据呢？如何衡量结果的正确性呢？在失败的情况下，你如何跟踪是哪个特定模块造成了失败呢？

这些问题都是可测试性设计(DfT)范式的基础。任何充分考虑了 DfT 原则的软件其本身都是可测试的，作为一个令人愉快的附属产物，它也是易于阅读和理解，可有助于以后日常维护的。

重要提示：

从我个人来看，可测试性远比测试本身要重要得多。可测试性是软件的一个特性，它代表了与软件质量有关的一个(重大的)声明。测试是一个旨在验证代码是否满足期望的过程。运用可测试性(例如，让你的代码易于测试)就像是学习捕鱼；编写单元测试就像是吃鱼。

9.1.1 DfT

DfT 作为一个普遍概念诞生于几十年前一个非软件行业的领域。事实上，DfT 的目的是为了改善主板和芯片中底层电路的生成过程。

DfT 先驱采用了很多设计技术和实践应用，旨在实现以自动化方式进行的有效测试。被先驱们称作“自动检测的设备”只不过是一组特设软件程序的集合，用来测试一些众所周知的主板功能并报告结果以用于诊断目的。

DfT 适应了软件工程，通过量身定制的程序被应用到代码的测试单元上来。最终，编写单元测试就像是编写软件。当你编写常规代码时，通常叫做类和函数，你关注的是程序的总体作用和实际的用例执行。相反，当你编写单元测试时，你更需要关注各个方法和类的输入输出，这是一个不同的粒度级别。

DfT 定义了任何软件单元都必须易于测试的三个特性：控制性、可见性和简单性。你会惊讶地发现这些特性解决的正是前面在讨论 DfT 的基础时所描述的问题。

1. 控制性的特性

控制性的特性指的是代码能够让测试人员将固定的输入数据应用到正在测试的软件中的程度。任何一款软件在编写时都应该明确需要哪些参数以及生成什么样的返回值。另外，任何软件都应该抽象其依赖性——参数和低层模块——并提供外部调用方可随意注入它们的方式。

应用于软件的控制性特性，一个典型示例是请求参数的方法，而不是利用对系统的了解从另一个可公开访问的组件找出参数的值。在 DfT 中，控制性主要是关于定义一个用于软件组件的虚拟协议，该组件中包括先决条件。配置先决条件越容易，就越容易编写有效的测试。

2. 可见性的特性

可见性的特性被定义为观察软件在测试状态下的当前状态和它可以产生输出的能力。实现了在方法上施加特设的输入值后，下一步就是验证方法是否如预期那样表现了。可见性主要是关于这一方面——在方法执行之后要验证的后置条件。

与可见性有关的主要假设是，如果测试人员能够以编程方式观察一个指定的行为，就能很容易地测试它不符合预期的地方或不正确的值。后置条件是将软件模块的预期行为形式化的一种方式。

3. 简单性的特性

简单性对于任何系统且在每个上下文中都是积极的特性。显然测试也不例外。简单且极为内聚的组件是最适合测试的，因为你需要测试的越少，测试过程就越迅速、可靠。

归根结底，DfT 是编写源代码的一个驱动因素，最好是从项目一开始编写源代码的时候就开始引入，这样可见性、控制性和简单性的特性就能够最大化。当成功应用了可测试性的设计时，编写单元测试总体上会变得简单高效。此外，你的代码最大化了可维护性，而且整体上更易于阅读了。

注意：

许多人都会同意，可维护性是软件需要专注的一个方面，因为它可以提供长效的好处。然而，可读性是与可维护性严格相关的，很大程度上，也是可维护性要努力的一部分。可读性是指编写易于阅读的代码，这样也就易于理解，在更新和改进时也更安全。可读性要通过全公司范围的命名和编码约定，并且更好的是，实现了把这些约定有效地传达给开发团队的方式。在这方面，Visual Studio Team Foundation Server 中的自定义策略能提供很大的帮助。

9.1.2　松散设计

从设计的角度看，可测试软件本质上是更好的软件。当你把控制性、可见性和简单性应

用于软件开发过程中时，最终会获得只通过约定接口就能进行交互的相对较小的构建块。编写可测试软件是方便其他人以编程方式使用它。可测试软件的典型编程用户是测试工具——用于运行单元测试的程序。在任何时候，我们谈论的都是会使用其他软件的软件。因此，低耦合是系统应用的普遍原则，遵循基于接口的编程是创建易于测试的软件的最好做法。

1. 基于接口的编程

紧耦合使软件开发变得更加简单快速。紧耦合产生于明显的一点：如果需要使用一个组件，只会得到它的一个实例。这会产生如下面所示的代码：

```
public class MyComponent
{
    private DefaultLogger _logger;
    public MyComponent()
    {
        _logger = new DefaultLogger();
    }
    public bool PerformTask()
    {
        // Some work here
        bool success = true;
        <...>

        // Log activity
        _logger.Log(...);

        // Return success or failure
        return success;
    }
}
```

MyComponent 类完全依赖于 DefaultLogger。你不能在没有 DefaultLogger 的环境中重用 MyComponent 类。此外，你不能在阻止 DefaultLogger 正常工作的运行时环境中重用 MyComponent 类。这是类之间紧耦合会带来问题的一个例子。从测试的角度看，不重建完全兼容生产环境的运行时环境，就不能测试 MyComponent 类。例如，如果 DefaultLogger 登录到微软互联网信息服务(IIS)，你的测试环境就必须正确配置 IIS 并使其正常工作。

另一方面，单元测试的好处是你能够快速、按时地运行测试，专注于软件一小部分的性能表现，忽略或控制依赖性。如果打算将类编程为使用依赖性的具体实现，明显就不可能做到了。下面介绍了如何重写 MyComponent 类，使它依赖于接口从而产生更多的可维护和可测试的代码。

```
public class MyComponent
{
    private ILogger _logger;
    public MyComponent()
    {
        _logger = new DefaultLogger();
    }
    public MyComponent(ILogger logger)
    {
        _logger = logger;
    }
    public bool PerformTask()
    {
        // Some work here
        bool success = true;
        ...

        // Log activity
        _logger.Log(...);

        // Return success or failure
        return success;
    }
}
```

MyComponent 类目前依赖于 ILogger 接口，该接口会抽象出日志记录模块的依赖性。MyComponent 类现在就清楚如何处理实现 ILogger 接口的任何对象了，包括你可能以编程方式注入的对象。

上面的解决方案从测试的角度看是可以接受的，即使它远非完美。在前面的实现中，MyComponent 类仍然依赖于 DefaultLogger，且在没有定义 DefaultLogger 的程序集的情况下你不能真正地重用它。不过，至少可以绕过默认的日志记录器用它单独测试 MyComponent 类的性能，如下所示：

```
// Arrange the call
var fakeLogger = new FakeLogger();
var component = new MyComponent(fakeLogger);

// Perform the call and check against expectations
Assert(component.PerformTask());
```

指示你的类面向接口运行而不是面向实现运行是现代软件开发的五大支柱之一。开发的五项原则常常被总结为首字母缩写的 SOLID 一词，下面是这五项原则首字母形式的含义：

- 单一职责原则
- 开闭原则
- 里氏替换原则
- 接口隔离原则
- 依赖反转原则

关于五项原则的更多信息，请查阅由我和 Andrea Saltarello 合著的 *Microsoft .NET Architecting Applications for the Enterprise*(微软出版社，2008 年出版)一书。

在现代软件中，编写面向接口而非面向实现的代码这一概念已被广为接受和应用了，但它也常常被另一个更具体的概念所掩盖：依赖注入。

我们可以说基于接口编程的整个概念是在依赖反转原则(DIP)中进行硬编码，且依赖注入是应用 DIP 的一种流行设计模式。第 7 章“设计 ASP.NET MVC 控制器的注意事项”中讨论了 DIP 和相关的模式，如依赖注入和服务定位器。

2. 软件可测试性的相对性

可测试性的设计很重要是因为它会产生易于测试的软件吗？或者说，是因为它会产生本质上具有更好设计的软件吗？我绝对赞成第二种说法(即使第一种说法也有强有力的论据)。

你不可能向客户推介可测试性特点来销售产品。你大概会专注在其他特征的推介上，比如功能、整体品质、用户友好度以及易用性。可测试性主要是对于开发人员很重要，因为这是设计和编码质量的晴雨表。从客户的角度来看，也许“能运行的可测试代码”和“能运行的不可测试代码”之间并没有什么区别。

另一方面，一款易于测试的软件一定是松散耦合的，它会提供核心部件之间的关注分离(SoC)，并且易于维护，因为它有一系列的测试能及时承载任何回归。除此之外，它的结构本身也更简单，能很好地适用于未来的扩展。

归根结底，追求可测试性是希望拥有具有良好设计软件的一个理由。此外，一旦你得到它，也能很容易地测试它！

注意：

我说过，从客户的角度看也许“能运行的可测试代码”和“能运行的不可测代码”之间没有区别。好吧，两者之间是否有差异实际上取决于客户。如果客户与你达成的是一个长期项目，他可能会对代码的可维护性很感兴趣。在这种情况下，代码是否具有可测试性就会产生很大的差别。但要再次说明的是，这更多涉及的是设计方面的内容而不是可测试方面的内容。

3. 可测试性与耦合

耦合与可测试性之间有着密切的关系。不能在测试中轻松实例化的类肯定有某些严重的耦合问题。这并不是说你不能对其自动测试，但你也许要在测试环境中配置一些数据库或外部连接，这就必然导致测试变慢，相对的维护成本也会增加。

有效的测试必须快，并在内存中执行。一个有良好测试覆盖率的项目很可能每个类都有几个简单的测试，加起来可能会有几千个测试调用。如果每个测试都足够快，且没有因同步和连接所导致的延时，那么这还是一个容易管理的问题。否则，问题就会变得比较严重。

如果组件之间的耦合问题未在设计中妥善解决，那么你最终还要测试与其他组件进行交互的组件，这看起来就更像是一个集成测试而非单元测试了。集成测试也是必要的，但理想情况下它们应该在独立的代码单元(比如类)上运行，而这些代码单元应该已经彻底进行过隔离测试了。集成测试不像单元测试进行地那么频繁，因为它们速度相对较慢，设置成本也更高。

另外，如果最终选用集成测试来测试一个类且出现了失败，那么如何才能够轻松地识别出问题所在呢？它是发生在你要测试的类中还是由某些依赖性所引发的呢？找到正确的问题会变得更加耗时耗力。甚至于即便你找到问题，对其进行修复还可能对上层组件产生不利影响。

要在设计层面保持对耦合的控制(比如系统性应用依赖注入)，可以强制执行可测试性。相反，通过追求可测试性，保持了对耦合的控制，最终就能拥有一个对软件的更好设计。

4. 可测试性与面向对象

一个主要争论点是为可测试性牺牲某些设计原则(特别是面向对象原则)是否值得(如果值得，要做出何种程度的牺牲)。如前所述，可测试性是追求更好设计的驱动力，但如果没有单元测试，你确实也可以有一个伟大的设计，同时有一个几乎不可能自动测试的伟大软件。

这里的要点略有不同。如果追求面向对象的优秀设计，可能会有一个策略，将虚拟成员和继承类的使用限制到只在绝对必要时使用。然而，非虚方法和密封类很难测试，因为大多数的测试环境都需要模拟类和重写成员。还有，你为什么要有一个除了测试别无他用的额外构造函数呢？你又应该做什么呢？

显然这是一个需要考虑和权衡的问题。

但是，你要考虑到商业工具的存在可以让你无视类的设计，对其进行模拟和测试，包括密封类和非虚方法。一个极好的例子是 Typemock (http://www.typemock.com)。很长一段时间，微软都有一个称为 Moles 的框架，它包含在 Visual Studio 2010 Power Tools (http://tinyurl.com/b4zuqj)中。Moles 不再受 Visual Studio 2013 的支持，它被一个新的称为 Fakes(http://bit.ly/zenveW)的框架所取代了。

在我们终于要处理 ASP.NET MVC 中的测试以前，先简要地回顾一下单元测试的基本知识。如果已经熟知像伪造、模拟、隔离测试这样的概念，则可以直接跳到 9.3 节。

9.2　单元测试的基本知识

单元测试会验证单独的代码单元是否按照它们预期的作用正常运行。单元是应用程序可测试的最小部分——通常是类上的一个方法。

单元测试包括编写和运行一个小的程序(称作之前提到的测试工具)，以自动的方式实例化测试类以及调用测试方法。测试类是一个承载测试方法的容器，而测试方法是通过使用一组特定的输入值从而调用方法进行测试的一个帮助器方法。归根结底，运行一系列的测试很像是在编译(参见图 9-1)。在选择的编程环境(例如 Visual Studio)中单击一个按钮，运行测试工具，在运行结束时，如果有错误，就会知道出了什么错。

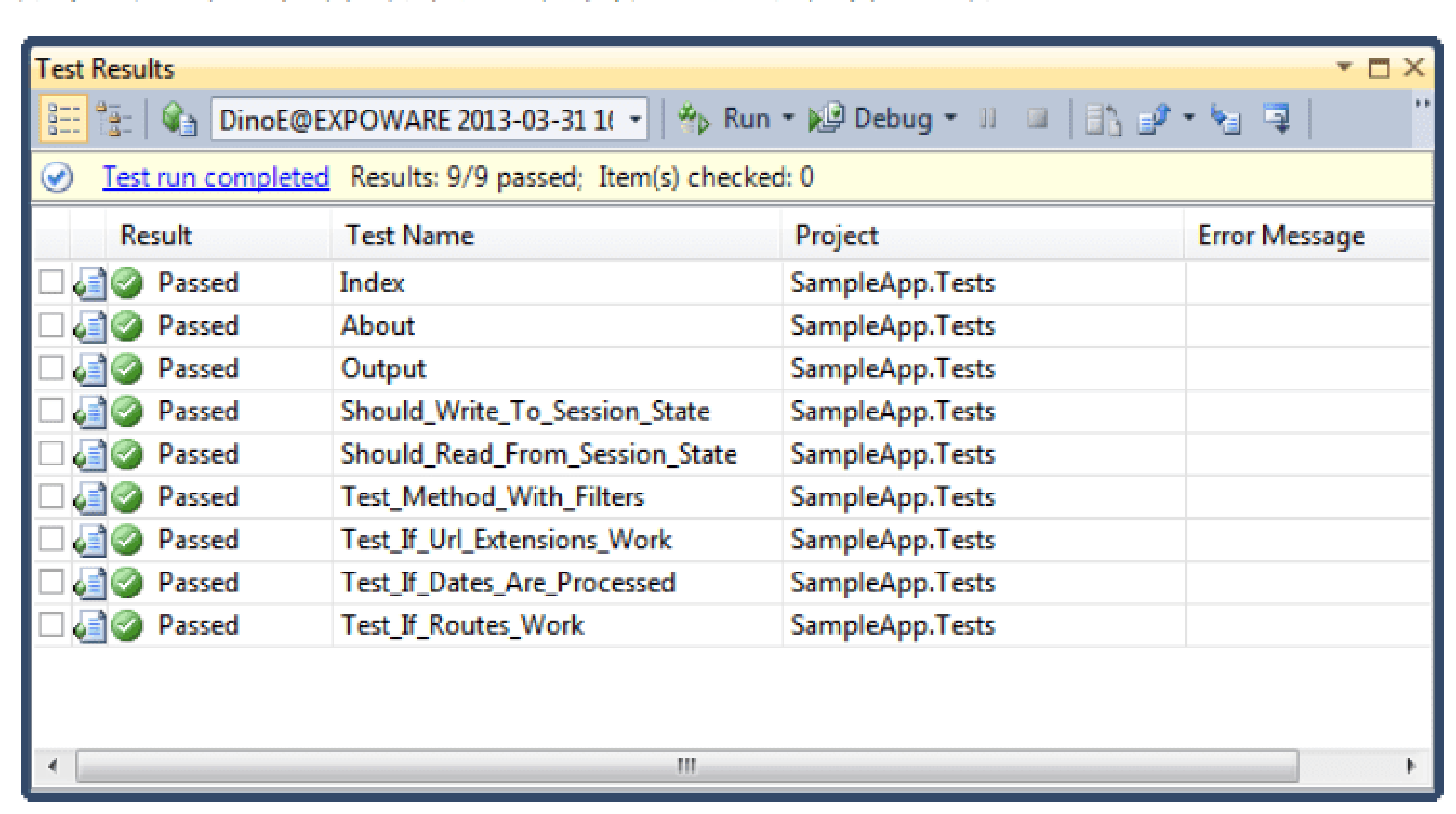


图 9-1　在 Visual Studio 中运行测试项目

9.2.1　使用测试工具

测试工具的最简单形式是一个手工编写的程序，它会从一些外部文件中读取测试用例的输入值和相应的预期结果。然后测试工具通过使用输入值来调用方法，并将结果与预期值进行比较。很明显，完全从头开始编写此类测试工具，至少是耗时且容易出错的。更重要的是，它在可以利用的测试功能方面是受限制的。

执行单元测试的最有效和最常见的方式是使用一种自动化的测试框架。自动化的测试框架是一个开发人员工具，通常包括一个运行时引擎和一个类的框架，用于简化测试程序的创建。

1. 选择一个测试环境

常用的测试工具是 MSTest、NUnit 及其继任者 xUnit.net。MSTest 是集成在 Visual Studio 所有版本中的测试工具。图 9-1 展示了 Visual Studio 中 MSTest 的用户界面。

NUnit(可以在 http://www.nunit.org 找到)是一款开源产品，已经面世好几年。NUnit 被设计为一个独立的工具，并不以本地方式与 Visual Studio 集成，这可以是好消息也可以是坏消息，取决于你看待它的角度以及你的需求和期望。不过，Visual Studio Gallery 中有一个集成包，以便在 Visual Studio 内部使用 NUnit。

xUnit.net(更多相关内容可以在 http://xunit.codeplex.com 中找到)是用于单元测试.NET 项目的最新框架。如图 9-2 所示，它配备了一个安装程序，用这个安装程序，可以将 xUnit.net 作为一个新的测试框架添加到创建新 ASP.NET MVC 应用程序的默认向导中。

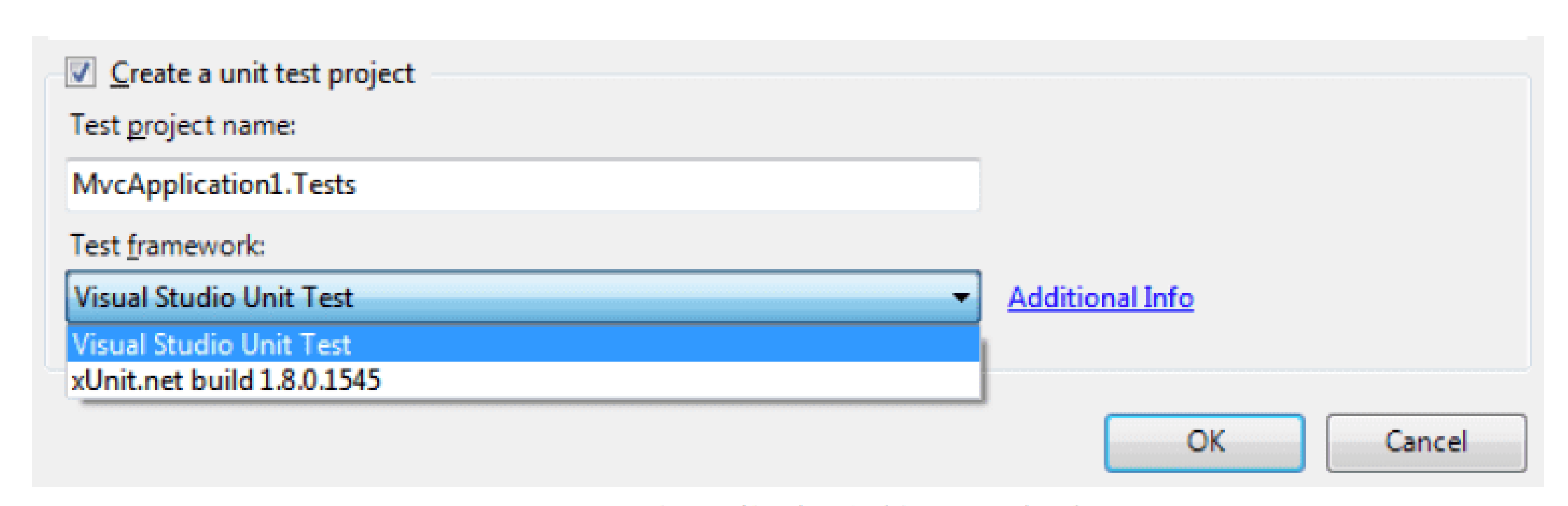


图 9-2　选取你喜欢的测试框架

最终，测试框架的选择还真的是见仁见智的问题。无论选择哪一个，客观上你几乎都不会失去什么真正重要的东西。但测试问题远不止所使用的框架那么简单。在我看来，MSTest 和 NUnit 之间没有太大的技术差别。那么 XUnit.net 呢？如前所述，xUnit.net 是最新的测试工具，它由 NUnit 的创始人 James Newkirk 所创建。基于这些原因，xUnit.net 很可能吸收了 NUnit 多年的经验和反馈，你也许会发现它更接近于你心目中的理想工具。

总体而言，测试框架在功能方面具有相似性并不意味着你就不能对它们有所偏好。然而，无论选择的理由是什么，它更多涉及的是个人喜好而非工具本身功能上的差异。这本书中我使用的是 MSTest，但我会简要指出与其他工具的差别，尤其是与 xUnit.net 的差别。

2. 测试固件

测试是从在测试固件中对相关测试进行分组开始的。测试固件是特定于测试的类，其中的方法通常代表了要运行的测试。在测试固件中，你可能还有在测试伊始和测试运行结束时执行的代码。下面是 MSTest 测试固件的主要结构：

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
...

[TestClass]
```

```
public class CustomerTestCase
{
  private Customer customer;

  [TestInitialize]
  public void SetUp()
  {
    customer = new Customer();
  }

  [TestCleanup]
  public void TearDown()
  {
    customer = null;
  }

  // Your tests go here
  [TestMethod]
  public void ShouldComplainInCaseOfInvalidId()
  {
    ...
  }
  ...
}
```

测试固件是在一个特设的 Visual Studio 项目中分组的。当你新创建了一个 ASP.NET MVC 项目时，Visual Studio 即为你提供一个新创建的测试项目。

通过简单地添加 TestClass 特性就可以把一个普通的.NET 类转变成一个测试固件。而通过使用 TestMethod 特性可以把.NET 类的方法转变成测试方法。TestInitialize 和 TestCleanup 这样的类具有特殊含义，分别表示.NET 类中需要在每个测试之前与之后运行的代码。而通过使用像 ClassInitialize 和 ClassCleanup 这样的特性，你就可以定义某个类中在每个测试之前和之后只运行一次的代码。

注意：

关于测试类(或固件)，在 xUnit.net 和其他工具之间存在一些差异。在 xUnit.net 中，你不需要用特殊的特性来修饰测试类。该框架会在所有可用的公共类中寻找所有的测试方法。就初始化和清理代码而言，xUnit.net 要求你将类构造函数和 Dispose 方法用于每个测试的任何操作。在你的测试类中要一次性实现 IUseFixture，还要实现类级别的初始化和拆卸。

3. 设置、作用、断言

测试方法的典型设计可总结为三个缩写的“A”，分别代表着设置、作用、断言。首先要设置上下文的执行，在其中要通过初始化类的状态和提供必要的依赖性来测试这个类。

接着，要将作用于该类的代码置于测试状态，并执行任何所需的任务。最后，处理结果并验证接收的输出是否正确。这一步是通过根据你的预期进行断言验证来实现的。

通过使用特设的由测试工具提供的断言 API 来编写你的断言。至少，有了测试框架，可以检查结果是否符合预期值。

```
[TestMethod]
public void AssignPropertyId()
{
   // Define the input data for the test
   var customer = new Customer();
   string id = "IDS";
   string expected = id;

   // Execute the action to test.
   customer.ID = id;

   // Test the results
   Assert.AreEqual(expected, customer.ID);
}
```

测试并不一定要检查结果是否正确。一个有效的测试也旨在验证在一定条件下某种方法是否会引发异常。下面是一个示例，如果分配了空字符串，Customer 类中 ID 属性的设置器就会引发 ArgumentException 异常：

```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void AssignPropertyId()
{
   // Define the input data for the test
   var customer = new Customer();
   var id = String.Empty;

   // Execute the action to test.
   customer.ID = id;

}
```

编写测试时，可以决定暂时忽略掉某个部分，因为你知道它不能正常运行，只是目前没有时间去修复。这种情况就可以使用如下所示的 Ignore 特性：

```
[Ignore]
[TestMethod]
public void AssignPropertyId()
{


}
```

同样，可以决定将测试标记为暂时不确定，因为你目前无法确定在哪些条件下测试会成功或者失败。

```
[TestMethod]
public void AssignPropertyId()
{
   ...
   Assert.Inconclusive("Unable to determine success or failure");
}
```

你也许会认为忽略掉一个测试或将其标记为不确定是多余的任务，因为可以直接将出于某些原因不能工作的测试给注释掉。这当然是可行的，但经验告诉我们测试是一项脆弱的任务，总是会在正常优先级和低优先级的界限之间摆动。此外，当它被注释掉以后会很容易被遗忘。所有的测试框架都提供了在保持代码活跃于项目的同时以编程方式忽略测试，这并不是偶然的。测试工具的作者知道项目进度和预算通常都很紧，但是他们也知道将测试维持在可执行状态也很重要。任何时候运行测试，你都会想起一些测试被忽略了或者是不确定的。总体来看，这优于仅仅把测试注释掉(同时也容易遗忘掉)。

注意：

xUnit.net 用一个新的 Assert 上的方法——Assert.Throws 方法——替换了 ExpectedException 特性。Ignore 被 Fact 特性(等同于 TestMethod 特性)上的 Skip 参数所取代。最后，Assert.Inconclusive 在 xUnit.net 中已不受支持了。要获知更多有关 xUnit.net 和其他框架之间差别的信息，请参见 http://xunit.codeplex.com/wikipage?title=Comparisons&referringTitle=Home。

4. 数据驱动的测试

当你为某个类方法安排一次测试时，有时可能需要尝试多个可能的值，包括正确的和不正确的值，以及表示边界条件的值。在这种情况下，数据驱动的测试是大有助益的。

MSTest 支持两个可能的数据源：微软 Office Excel 的.csv 文件和任何有效的 ADO.NET 数据源。测试必须通过使用 DataSource 特性绑定到数据源，并运行一个测试实例用于数据源中的每个值。数据源会包含输入值和预期值。

```
var id = TestContext.DataRow["ID"].ToString();
var expected = TestContext.DataRow["Result"].ToString();
...
Assert.AreEqual(id, expected);
```

TestContext 变量用于读取输入的值。在 MSTest 中，当你添加一个新的单元测试时，TestContext 变量会自动定义：

```
private TestContext testContextInstance;
public TestContext TestContext
{
   get { return testContextInstance; }
   set { testContextInstance = value; }
}
```

尤其是，DataSource 特性还允许你指定测试输入值的顺序，是随机处理还是按顺序处理。

9.2.2　测试的特性

编写单元测试仍然是一种编程，与生产代码的软件编程一样，也需要良好的实践和技术。但是单元测试的编写还有其自己的一组模式和特点，你必须要了解。

1. 非常有限的范围

在本章开头介绍 DfT 的时候，我就特意提出了以下观点：简单性是软件的一个基本方面，是启用可测试性的关键。当应用到单元测试时，简单性就涉及测试指定的有限范围内的代码。

有限的范围使测试不言自明并清楚地揭示了其目的。至少有两个原因说明这样做是有益的。第一，任何检查该测试的开发人员，包括几星期之后再回来检查的原编写者，都可以快速、明确地理解被测试方法的预期行为。

第二，一个失败的测试会带来额外的麻烦，你需要找出失败的原因以便修复被测试的类。测试方法越简单，就越容易在被测试的类中将问题隔离出来。此外，被测试类的分层越多，就越容易应用更改而没有破坏别处代码的风险。最后，编写小范围的测试对于将依赖性控制在其他组件的类来说更为容易。

单元测试就像一个闭环：它是良性还是恶性完全取决于你，而且主要取决于设计的质量。

2. 隔离测试

与小范围单元测试紧密相关的一个方面是测试隔离。在测试一个方法时，你会希望将注意力放在该方法中的代码上。你需要知道的只是该代码在测试场景中是否提供了预期的结果。要获知这个信息，就需要摆脱该方法可能有的所有依赖性。

例如，如果该方法调用另一个类，你就要假定被调用的类始终会返回正确的结果。这样，

你就从根源上消除了该方法测试失败的风险，因为失败是沿着调用堆栈发生的。如果测试 A 方法失败了，就可以特定在 A 方法的源代码中找原因而不是在 A 方法的任何依赖性中找。

强烈建议你把正在测试的类从其依赖性中隔离开来。但是要注意，这只在此类是以松散耦合的方式设计的情况下可行。在面向对象的场景中，当验证了下面任何一个条件时，A 类都依赖于 B 类：

- A 类派生于 B 类。
- A 类包含 B 类的成员
- A 类的方法之一调用了 B 类的方法
- A 类的方法之一接收或返回 B 类的参数
- A 类依赖于一个类，这个类又依赖于 B 类。

如何在测试一个方法时消除依赖性呢？请使用测试替身。

3. 伪造和模拟

测试替身(test double)是用来代替另一个对象的对象。测试替身用来假装成指定场景真正期望的那个对象。使用一个实现 ILogger 接口的对象的类可以接受真正的记录到 IIS 或某些数据库表的记录器对象。同时，它还可以接受一个假装是记录器但不执行任何操作的对象。有两种主要类型的测试替身：伪造和模拟。

最简单的选择是使用伪造对象。伪造对象是一个对象的相对简单的克隆，它提供与原始对象相同的接口，但返回硬编码或以编程方式确定的值。下面是一个 ILogger 类型的伪造对象示例：

```
public class FakeLogger : ILogger
{
    public void Log(String message)
    {
        return;
    }
}
```

可以看出，伪造对象的行为是硬编码的；伪造对象不具有状态和明显的行为。从伪造对象的角度来看，你调用一个伪造方法的次数以及该调用发生在流中的时间都没有区别。在想要忽略依赖性的时候就可以使用伪造方法。

一个更复杂的选项是使用模拟对象。伪造对象能完成的任务模拟对象都能完成，但模拟对象还能完成更多的任务。在某种程度上，一个模拟就是具有它自己特性的对象，这些特性需要模拟另一个对象的行为和接口。

一个模拟还能向测试人员提供什么呢？基本上，模拟可适应对方法调用上下文的验证。借助模拟，你就可以验证方法调用是否是在合适的前提条件下、以该类中与其他方法有关的

正确顺序发生的。

手动编写一个伪造通常不是什么大问题；大多数情况下，你需要的所有逻辑都简单且不需频繁更改。当你使用伪造时，主要关注的是伪造对象可能代表的状态；而不是想与它交互。相反，当你需要在测试过程中与依赖对象进行交互时要使用模拟对象。例如，你也许想知道模拟对象是否被调用，并且可能要决定测试中对于指定方法必须返回什么样的模拟对象。

手动编写模拟当然是可行的，但它很少成为你会考虑的选项。从你期望从一个模拟中获得的灵活性程度考虑，你需要的是一个特设的模拟框架。表 9-1 列出了几个流行的模拟框架。

表 9-1　一些流行的模拟框架

产　　品	URL
Moq	http://code.google.com/p/moq
NMock2	http://sourceforge.net/projects/nmock2
Typemock	http://www.typemock.com
Rhino Mocks	http://hibernatingrhinos.com/open-source/rhino-mocks

注意，目前没有模拟框架被集成在 Visual Studio 及其早期的版本中。

除了 Typemock 是个明显的例外，表中的其他框架都是开源软件。Typemock 是具有独特功能的商业产品，基本上不需要为可测试性(重新)设计你的代码。使用 Typemock，可以测试之前被认为是不可测试的方法，如静态方法、非虚拟方法和静态类。

下面是如何使用像 Moq 这样的模拟框架的一个快速示例：

```
[TestMethod]
public void Test_If_Method_Works()
{
    // Arrange
    var logger = new Mock<ILogger>();
    logger.Setup(l => l.Log(It.IsAny<String>()))
    var controller = new HomeController(logger);

    // Act
    ...

    // Assert
    ...
}
```

被测试的类(HomeController 类)在实现 ILogger 接口的对象上有一个依赖性：

```
public interface ILogger
```

```
{
    void Log(String msg);
}
```

mock 存储库提供了一个动态创建的对象，它会模拟出测试将会使用的接口。模拟对象实现 Log 方法的方式是不对该方法接收到的任何字符串参数做处理。你并不真的在测试记录器；而是专注于控制器类，为控制器内部所使用的记录器组件提供快速实用的模拟。

你没有必要创建一个完整的伪造类；只需要指定在调用时给定方法运行所需的代码。这就是模拟相对于伪造的能力。

4. 每个测试中断言的数量

每个测试应该有多少断言呢？为了遵循小范围测试的准则，你就必须强制把每个测试限定为一个断言吗？这是一个有争议的地方。

业界不少人似乎都认为应该如此。事实上，支持这一观点的论据也相当好。每个测试一个断言会引导你编写更专注的测试，并保持你的限定范围。每个测试一个断言会使每个测试的测试目的一目了然。

多个断言的需求常常会掩盖你在单个测试中测试很多功能的事实。这显然是要避免的事情。如果只能选择一个原则去遵循，每个测试一个断言的原则可能是最好的。如果在指定操作后测试某个对象的状态，则可能需要检查多个值，并需要多个断言。你当然也可以通过一连串的测试来表示这些内容，每个测试有一个断言。但是在我看来，那需要大量的重构，而获益不多。

我不介意在一个测试中有多个断言，只要测试中的代码测试的是非常特定的性能。大多数框架都会止于第一个失败的断言处，所以从理论上讲你的风险是，同一个测试中的其他断言在下一次运行时也可能会失败。如果坚守原则只测试一种行为，并使用多个断言来验证与该行为相关的类的多个方面，那么所有的断言就都是相关的，并且如果第一个失败，通过修复它有很大可能你不会在该测试中得到更多的失败。

5. 测试内部成员

在某些情况下，受保护的方法或属性需要在测试中访问。一般而言，类成员不一定要公开才能得到测试。但是，测试非公共成员却会带来别的问题。

测试非公共成员的常用方式是创建一个扩展所测试类的新类。然后派生的类要添加一个公共方法来调用受保护的方法。添加这个类只是用于测试项目，并不会破坏类的设计。

正如前面提到的，.NET 框架中更好的方式是添加一个所测试的类的部分类。但是要实现这一目标，原始类自己也需要被标记为部分。不过，从设计方面来看，这不是什么大问题。

在.NET 中，也可以轻易地通过使用 InternalsVisibleTo 特性，使一个类的内部成员可见于

另一个程序集(比如测试程序集)，如下所示：

```
[assembly: InternalsVisibleTo("MyTests")]
```

可以将上面的这行代码添加到项目的 assemblyinfo.cs 文件中，该项目包含了带有内部成员的那个类，这样就能使其可用了。请注意可以多次使用这个特性，以便让类的内部成员对多个外部可执行程序可见。

依我之见，使用此特性比使用部分类要更冒失一些。事实上，要利用该特性，必须将你希望从测试中收回的任何成员标记为内部成员。内部成员仍然没有被公开可用，但它们的可见级别已经比非公开或受保护要高了。换句话说，你应该尽量少使用内部成员和 InternalsToVisible，并且仅在有具体需要时才使用。

最后，MSTest 还有一个不错的编程功能——PrivateObject 类——它提供了通过反射调用非公共成员的功能。

```
var resourceId = "WelcomeMessage";
var resourceFile = "MyRes.it.resx";
var expected = "...";
var po = new PrivateObject(controller);
var text = po.Invoke("GetLocalizedText", new object[] { resourceId,
           resourceFile });
Assert.AreEqual(text, expected);
```

你要将包含隐藏成员的对象封装在 PrivateObject 类的一个新实例中。接着，要调用 Invoke 方法以间接调用带有作为参数列表的对象数组的那个方法。Invoke 方法会返回一个表示私有成员返回值的对象。

6. 代码覆盖率

单元测试和集成测试的主要目的是使开发团队对他们生产的软件质量有信心。基本上，单元测试能够告知开发团队他们做得好不好，是否在正确的轨道上。但是，单元测试结果的可靠性到底有多高呢？

对于可靠性的衡量在某种程度上还取决于单元测试的数量和测试所覆盖的代码比例。换句话说，并不能证明代码覆盖率与软件质量之间存在着实际的相关性。

通常情况下，单元测试只会覆盖代码库的一个子集，对于代码覆盖比例达到多少算好并没有一个统一的定论。有人说 80%比较好；有些人甚至不愿给出具体数字。可以肯定的是，完全的代码覆盖率实际上是无法实现的，即使理论上有这种可能。

所有版本的 Visual Studio 都有代码覆盖率工具。

另外，代码覆盖率是很通用的术语，可以涉及很多不同的计算条件，如函数、语句、判定和路径覆盖。函数覆盖测量的是项目中的每个函数是否都已经在测试中得到了执行。语句

覆盖更细致地着眼于源代码的独立行。判定覆盖测量的是分支(如 if 语句)是否被评测，而路径覆盖会检查是否执行了每一个通过指定代码部分的可能路径。

每个标准都提供了一个看待代码的视角，但返回的仅仅是待说明的数字。因此，测试所有的代码行(即 100%的语句覆盖率)看起来是一件很棒的事情；然而，具有更高值的路径覆盖率也许更可取。代码覆盖率当然很有用，因为它可以帮助你识别哪些代码还没有被测试。然而，代码覆盖率并不会使你获知很多与测试对代码的执行情况有关的信息。举一个能很好说明这一情况的例子吧。

想象一个处理整数的方法。可以对它进行 100%的语句覆盖，但如果缺少了一个该方法接收了超出范围的无效值的测试，则可能在运行时会出现异常，尽管你已运行了前面所有成功的测试。

归根结底，代码覆盖率是一个有关特殊测量的数字问题。测试的关联性是真正重要的事情。盲目地增加代码覆盖率，或更糟的是，要求开发人员达到指定的覆盖率阈值并不能保证任何事情。这比没有测试可能还是要好一些，但它并不能说明测试的相关性和有效性。专注于预期的行为和预期的输入是达成测试目标的最合理方式。一个经过良好测试的应用程序就是对相关应用场景具有高覆盖率的应用程序。

注意：

微软用于 Visual Studio 的 Pex 插件旨在理解你的代码逻辑并为你建议需要进行的相关测试。在内部，Pex 采用了静态分析技术，以构建有关你的应用程序行为的知识基础。可以从 http://tinyurl.com/b4zuqj 处下载 Pex。

9.3　测试 ASP.NET MVC 代码

可测试性常常被看作是使 ASP.NET MVC 成为微软平台 Web 开发的首要考虑选项的一个不可分割的功能。由于 ASP.NET MVC 有视图和行为之间的 SoC，所以确实有助于开发人员编写更为健壮和精心设计的软件。ASP.NET MVC 运行时还提供了一个 API，它会抽象出你的代码在 ASP.NET 内部对象上可能有的任何依赖性。就测试而言，这种改变标志着与 Web Forms 的巨大区别。最起码，ASP.NET MVC 绝对是一个便于单元测试的框架。

9.3.1　应该测试哪部分代码

如前所述，ASP.NET MVC 提供了应用程序的支柱——控制器、视图和模型之间的有序分离。此外，它减少了你的代码对 ASP.NET 运行时内部组件的依赖，比如 Request、Response 和 Session。模板项目还提供了一个 global.asax 文件，其中所有的初始化工作都是以面向测试的方式编写的；这些都是能起到帮助作用的一些小细节。

ASP.NET MVC 尽其所能地启用和支持了测试。然而，ASP.NET MVC 不会编写你的测试，对你应用程序的实际结构和构成层也一无所知。你的目标应该是编写相关的测试；而不应该简单地追求代码的高覆盖率。

1. 如何找到要测试的相关代码

要测试的相关代码的位置主要取决于代码中的层。根据第 7 章中所讨论的，我的意见是，“不要把它放在控制器中。”很多强调对 ASP.NET MVC 提供的单元测试进行支持的例子都仅限于测试控制器。请思考下面的代码：

```
[TestClass]
public class HomeControllerTest
{
    [TestMethod]
    public void Index()
    {
        var controller = new HomeController();
        var result = controller.Index() as ViewResult;
        Assert.AreEqual("Welcome to ASP.NET MVC!", result.ViewBag.Message);
    }
}
```

该测试创建了一个控制器类的新实例，并调用了 Index 方法。该方法会返回一个 ViewResult 对象。然后断言会检查 ViewResult 实例中的 Message 属性是否等于指定的字符串。让我们在此处回顾一下该控制器的代码：

```
public ActionResult Index()
{
    ViewBag.Message = "Welcome to ASP.NET MVC!";
    return View();
}
```

此代码的相关部分是任务分配。为了保持控制器的至精至简，你应该考虑将这段代码移到工作线程服务，如第 7 章中所讨论的。下面是如何重写该方法以隔离核心逻辑的代码：

```
public ActionResult Index()
{
    _service.GetIndexViewModel(ViewBag);
    return View();
}
```

此时，你不再需要测试控制器了。你也许会想要测试服务类。可能存在着控制器方法主体有点臃肿的情况；不过，大部分是由条件语句和琐碎任务构成的胶水代码——没什么真正

需要你测试的。

可以将同样的推论应用到工作线程服务，引领你为具有极其清晰和有限断言的高度专用组件编写测试。

注意：

在编写单元测试时，你需要知道很多有关你要测试单元的内部详细信息。事实上，单元测试是一种白盒测试，与黑盒测试相反，测试人员不需要了解内部信息，并且将测试限定于输入指定的输入值和期待指定的输出值。

2. 域层

第 7 章定义了域层的上下文，它涉及应用程序业务上下文的恒定对象——数据和行为。如果要设计一个电子商务应用程序，你的域会主要包括像发票、客户、订单、运输者以及报价这样的实体。上述每个实体都有属性和方法。例如，发票具有的属性有 Number、Date、Payment 和 Items 等，具有的方法有 GetEstimatedDayOfPayment、GetTotal 和 CalculateTaxes 等。对于这其中的一些实体(比如聚合根)来说，你可能还需要以跨实体方式执行特设操作的特殊服务。例如，你可能希望有一个方法能获取客户信息并计算出她是否下了足够多的订单从而成为金牌客户。

这是你绝对要彻底测试的一部分代码；也就是说，你想确保它被相关的单元测试广泛覆盖。因为这是你应用程序的核心，所以你会希望确保你遇到了所有适当的边界情况，并正确地检测出了不一致的值/状态。最后，你想确保一系列适当的测试可以提醒你注意在开发后期引入的所有回归。如果只能用测试覆盖应用程序的一部分的话，我建议这一部分是域层，如果有的话。

3. 编排层

取决于你真正实现的层数和层级数，编排层(在第 7 章中讨论过)既可以由 ASP.NET MVC 控制器的工作线程服务完全识别，也可以形成其自己的一层。

测试这一层的需求完全取决于你在其中所拥有的逻辑数量。在创建、读取、更新和删除(CRUD)系统中，这一层通常足够精简，只需要测试它的样本即可。然而，如果表示层提供了与存储保存的数据明显不同的数据表示形式，编排层则会负责以一种特定于视图的格式来安排数据。在这种情况下，这一层会成为你应用程序中更为重要的一部分，值得更加关注。

如果将控制器简化成一个纯粹的传递层——从编排层获取视图模型并将其传递到 ASP.NET MVC 基础架构——就没有必要对其进行测试。或者，更好的是，你首先就专注在应用程序的其他部分了。

4. 数据访问层

你的数据访问层提供了什么服务？它只是为你运行 SQL 语句吗？如果是这样的话，在你确定代码在开发时能正常运行(例如，你的 SQL 语句是正确的)之后，一切就都继续了。

如果数据访问层总结了额外的功能，并包含了一些将数据修改成与存储不同的数据结构的逻辑，你可能就需要考虑一些测试了。但是，在这种情况下，为什么不把 CRUD 内容从适配器外壳中分离出来，只测试适配器呢？

重要提示：

单元测试并不真的是一个数字问题；而是一个数字的质量问题。你不仅需要测试，还需要覆盖代码相关方面的测试。你不可能因为单元测试就能销售出应用程序；但会因为应用程序通过了验收测试而售出应用程序。而且，验收测试会表明何种行为是对最终用户相关的。找出哪些行为与构成你应用程序的代码单元有关，正是你期望从伟大的开发人员那里学到的东西。

9.3.2　对 ASP.NET MVC 代码进行单元测试

超出单元测试理论之外的——有人甚至称它为单元测试之艺术——还有一些具体而务实的方面需要你处理。具体来说，有一些实践和技术你需要理解以便编写用于 ASP.NET MVC 应用程序的单元测试。

编写单元测试相当于调用一个模拟你关注的特定上下文的方法。单元测试就是一个软件——一个类中的方法——有了它，就像软件一样，可以使用在常规代码类中所使用的大部分技巧和技术。在这一章中，我只讨论需要知道的相关方面，而非 ASP.NET MVC 单元测试的每一个方面。

注意：

如果对单元测试的艺术感兴趣，务必购买一本由 Roy Osherove 编写的著作 *The Art of Unit Testing*(*Manning Publications*，2009 年出版)，可以在 http://www.manning.com/osherove 找到。

1. 测试返回的视图是否正确

可能存在着控制器动态决定要呈现的视图的情形。当要呈现的视图是基于某些只能在运行时获知的条件时，就会发生这种情况。一个例子是，必须基于区域设置、用户账户、星期几、或任何用户可能要求的条件来切换视图模板的一个控制器方法。

在测试中，可以通过使用 ActionResult 对象的 ViewName 属性来获得正在呈现的视图，如下所示：

```
[TestMethod]
public void Should_Render_Italian_View()
{
   // Simulate ad hoc runtime conditions here
   ...

   // Parameters
   var productId = 42;
   var expectedViewName = "index_it";

   // Go
   var controller = new ProductController();
   var result = controller.Find(productId) as ViewResult;
   if (result == null)
      Assert.Fail("Invalid result");
   Assert.AreEqual(result.ViewName, expectedViewName);
}
```

假设 ProductController 类会为所选产品返回本地化视图。在这种情况下，运行时条件为了测试而模拟的一个好例子是将其设置为当前区域。

通过检查由控制器方法返回的特定 ActionResult 对象的公共属性，你还可以在生成特定响应——如 JavaScript 对象标记(JSON)、JavaScript、二进制数据、文件等——的时候执行特设检查。

2. 测试本地化

有时候你会发现，做一些快速检查的测试，看看在选择了某种特定语言时用户界面的某些部分是否会得到适当的本地化资源是非常有用的。下面是如何使用单元测试进行检查的代码：

```
[TestMethod]
public void Test_If_Localizated_Strings_Are_Used()
{
    // Simulate ad hoc runtime conditions here
    const String culture = "it-IT";
    var cultureInfo = CultureInfo.CreateSpecificCulture(culture);
    Thread.CurrentThread.CurrentCulture = cultureInfo;
    Thread.CurrentThread.CurrentUICulture = cultureInfo;

    // Ensure resources are being returned in the correct language
    var showMeMoreDetails = MyText.Product.ShowMeDetails;

    // Assert
```

```
    Assert.AreEqual(showMeMoreDetails, "Maggiori informazioni");
}
```

在单元测试中，首先要在当前线程上设置区域性，然后尝试检索资源值，并且根据预期值进行断言。

可以使用这一技术来测试与本地化相关的几乎一切内容，包括局部视图和资源。下面是为 UrlHelper 扩展方法而编写的单元测试，这在第 5 章“ASP.NET MVC 应用程序特性”中讨论过：

```
[TestMethod]
public void Test_If_Url_Extensions_Work()
{
    // Data
    var url = "sample.css";
    var expectedUrl = "sample.it.css";

    // Set culture to IT
    const String culture = "it-IT";
    var cultureInfo = CultureInfo.CreateSpecificCulture(culture);
    Thread.CurrentThread.CurrentCulture = cultureInfo;
    Thread.CurrentThread.CurrentUICulture = cultureInfo;

    // Act & Assert
    var localizedUrl = UrlExtensions.GetLocalizedUrl(url);
    Assert.AreEqual(localizedUrl, expectedUrl);
}
```

这一快速演示隐藏了一件非常有趣的事情。第 5 章显示了定义 GetLocalizedUrl 扩展方法的代码，如下所示：

```
public static String GetLocalizedUrl(UrlHelper helper, String resourceUrl)
```

要测试此方法，你需要提供一个 UrlHelper 类的实例。可惜的是，UrlHelper 类的构造函数是与 ASP.NET MVC 基础架构联系在一起的。

```
public UrlHelper(RequestContext context)
```

如何在测试环境中得到一个有效的 RequestContext 呢？你需要模拟出 HTTP 上下文。这绝对是可行的事，如同之后你将会看到的，但在这个方案中它需要太多的工作了。一个简单的重构可以帮助你专注于真正与测试相关的东西。

最后你需要的是检查当区域性是 Italian 时，代码返回 sample.it.css 而非 sample.css 的能力。你不需要测试代码是否真的存在于 Web 服务器上。因此，并不完全需要请求上下文。让我们

重写 GetLocalizedUrl 方法，如下所示：

```
public static String GetLocalizedUrl(UrlHelper helper, String resourceUrl)
{
    var url = GetLocalizedUrl(resourceUrl);
    return VirtualFileExists(helper, url) ? url : resourceUrl;
}
public static String GetLocalizedUrl(String resourceUrl)
{
    var cultureExt = String.Format("{0}{1}",
        Thread.CurrentThread.CurrentUICulture.TwoLetterISOLanguageName,
        Path.GetExtension(resourceUrl));
    return Path.ChangeExtension(resourceUrl, cultureExt);
}
```

其效果是一样的，但测试却更快且更具有针对性了。

3. 测试重定向

控制器操作可能也会重定向到另一个 URL 或路由。但是，测试重定向并不比测试特定于上下文的视图更复杂。如果控制器方法重定向到特定 URL 的话，则重定向会返回一个 RedirectResult 对象；然而，如果它重定向到指定的路由，就会返回一个 RedirectToRouteResult 对象。

RedirectResult 类具有一个熟悉的 Url 属性，可以检查以验证该操作是否成功完成了。RedirectToRouteResult 类具有如 RouteName 和 RouteValues 这样的属性，可以检查以确保重定向工作正常。

4. 测试路由

尤其在你广泛使用自定义路由的情况下，你可能会希望仔细地测试它们。具体来说，你关注的是检查指定的 URL 是否匹配到了正确的路由，以及路由数据是否被正确地提取。

要测试路由，你必须重建 global.asax 环境，并且首先调用 RegisterRoutes 方法。RegisterRoutes 方法填充了具有可用路由的集合，如以下所示：

```
[TestMethod]
public void Test_If_Product_Routes_Work()
{
    // Arrange
    var routes = new RouteCollection();
    MvcApplication.RegisterRoutes(routes);
    RouteData routeData = null;
    // Act & Assert whether the right route was found
```

```
    var expectedRoute = "{controller}/{action}/{id}";
    routeData = GetRouteDataForUrl("~/product/id/123", routes);
    Assert.AreEqual(((Route) routeData.Route).Url, expectedRoute);
}
```

测试中的 GetRouteDataForUrl 方法是一个本地帮助器，定义如下：

```
private static RouteData GetRouteDataForUrl(String url, RouteCollection routes)
{
    var httpContextMock = new Mock<HttpContextBase>();
    httpContextMock.Setup(c => c.Request.
                        AppRelativeCurrentExecutionFilePath).Returns(url);
    var routeData = routes.GetRouteData(httpContextMock);
    Assert.IsNotNull(routeData, "Should have found the route");
    return routeData;
}
```

该方法预计将调用 GetRouteData 来检索有关所请求路由的信息。可惜的是，GetRouteData 需要一个对 HttpContextBase 的引用，它会放置所有有关请求的查询。尤其是，GetRouteData 需要调用 AppRelativeCurrentExecutionFilePath 以了解要处理的虚拟路径。通过模拟 HttpContextBase 以提供特设的 URL，你就可以将路由从运行时环境中完全解耦出来并用断言进行处理。

前面所示的示例代码使用了 Moq 框架来创建测试替身。让我们进一步了解模拟以及如何利用模拟来抵消或替换依赖性。

9.3.3　处理依赖性

就测试而言，可以说有两个主要类型的依赖性：那些你想要忽略的，以及那些你想要与其交互但要以受控的方式进行的。在这两种情形下，你都需要提供一个测试替身对象——即在提供预期性能的同时表现得也像预期的对象。如果被测试的类支持依赖注入(DI)，提供测试替身就是小菜一碟了。

无论用来表示测试替身(伪造、模拟、存根)的名称是什么，纯粹的事实是你需要一个实现指定协议的对象。那么，如何编写一个这样的测试替身对象呢？

1. 关于模拟和伪造对象

测试替身是由你编写并添加到测试项目中的一个类。这个类实现指定的接口，或继承自一个指定的基类。在有了该实例以后，通过使用被测试对象的公共接口将其注入到受测对象的内部(很明显，我假定受测对象的设计考虑了可测试性要求)。

可能每一个你编写的测试都需要一个不同的测试替身。这些类看起来几乎相同，但它们所返回的值可能会不同。你真的需要编写和维护成百上千个类似的类吗？当然不用，这就是

模拟框架存在的理由。模拟框架为你提供了快速创建能够实现协议的类的基础架构。另外，模拟框架还提供了一些工具，可以用来配置与该动态创建的类方法的交互。尤其是，可以指示模拟获得一个公开 ISomething 协议的类的实例，并且在用指定参数调用 ExecuteTask 方法时返回 1。

当你需要在一个或多个方法的实现中保持某种状态或执行某些自定义逻辑时，大概会需要编写你自己的类。在这本书的上下文中，我将伪造称为你在测试项目中编写的专用于消除依赖性的类。我将模拟称为你使用模拟框架创建的用于相同目的的类。

你仍然认为伪造和模拟之间存在相关性差异吗？如果真的存在，也只是你希望附加给它们的标签罢了。

2. 执行数据访问的测试代码

DI 在测试中的典型例子是，当你有一个工作线程服务类(在简单的情形中甚至可以是控制器类)时，需要执行数据访问操作。第 7 章定义了一个从存储库中获取日期列表的名为 HomeService 的工作线程服务类。该服务类被用来在把日期列表封装进用于显示的视图模型之前对其做一些额外的工作。尤其是，工作线程服务会计算当前日期与指定日期之间的时间跨度。存储库可能会运行一些数据库查询以返回日期。下面是如何在服务类中注入伪造依赖性的代码，以便可以在不处理查询和连接字符串的情况下测试服务类：

```
[TestClass]
public class DateRepositoryTests
{
    [TestMethod]
    public void Test_If_Dates_Are_Processed()
    {
        var inputDate = new DateTime(2013, 2, 8);
        var fakeRepository = new FakeDateRepository();
        var service = new HomeServices(fakeRepository);
        var model = service.GetHomeViewModel();

        var expectedResult = (Int32) (DateTime.Now - inputDate).TotalDays;
        Assert.AreEqual(model.FeaturedDates[0].DaysToGo, expectedResult);
    }
}
```

FakeDateRepository 类看起来如下：

```
public class FakeDateRepository : IDateRepository
{
   public override IList<MementoDate> GetFeaturedDates()
   {
```

```
        return List<MementoDate>
                    {
                       new MementoDate {Date = new DateTime(...)}
                    });
    }
}
```

你打算编写的每个测试都需要一个新的 FakeDateRepository 类。例如，假设你希望在指定日期到来之前和当前日期之后都要测试服务行为。你需要两个测试以及两个版本略微不同的 FakeDateRepository，区别仅在于所返回的日期。下面的代码揭示了模拟框架是如何起到帮助作用的：

```
[TestClass]
public class DateRepositoryTests
{
    [TestMethod]
    public void Test_If_Dates_Are_Processed()
    {
        var inputDate = new DateTime(2012, 2, 8);
        var fakeRepository = new Mock<IDateRepository>();
        fakeRepository.Setup(d => d.GetFeaturedDates()).Returns
          (new List<MementoDate>
                                    {
                                       new MementoDate {Date = inputDate}
                                     });

        var service = new HomeServices(fakeRepository.Object);
        var model = service.GetHomeViewModel();

        var expectedResult = (Int32) (DateTime.Now - inputDate).TotalDays;
        Assert.AreEqual(model.FeaturedDates[0].DaysToGo, expectedResult);
    }
}
```

伪造存储库是通过使用 Moq 创建的，并通过构造函数注入 HomeService 类。Mock<IDateRepository>对象是一个动态创建的类(Moq 在内部使用 Castle DynamicProxy 来动态生成代码)，每当调用 GetFeaturedDates 方法的时候，它就会实现 IDateRepository 接口并返回输入的日期。要编写针对另一个输入的测试，你不需要显式处理源类，直接重复测试方法就行。

9.3.4　模拟 HTTP 上下文

当不得不模拟 HTTP 上下文以编写某些 ASP.NET MVC 单元测试时，让我感到不舒服。

有时候，你确实需要它；然而，更多的情况往往是，在你的代码上进行一点重构就行了，并没有必要模拟 HTTP 上下文。

在面对单元测试时，我经常会发现一个错误(至少我称它为错误)。一些开发人员似乎无法看控制器级别以外的内容。这些开发人员在控制器中放入了大量的逻辑。他们很少利用特定的、高度专用的类。这些开发人员似乎认为在控制器以外，你只能使用一个微软 SQL Server 数据库或实体框架模型。有一个围绕 LINQ-to-Entities 查询的存储库层就已经是一个前沿的解决方案了。第 7 章尝试了对纯代码的一种不同模式进行阐述。

如果控制器方法的粒度相当粗大，你便不可避免地需要模拟一长串运行测试的对象。直接处理查询字符串和会话状态的控制器方法需要有一个有效的 HTTP 上下文来安排要运行的测试。这是你的一项额外工作。另外，重构添加封装和包装器的控制器方法也是你的额外工作，但它是另一种类型的工作。

你很快就可以看到简洁设计的好处了：首先，测试变得更加容易。为让测试能正常工作会耗费很多精力，结果却是浪费了更多的时间。当测试最终开始运行时，你感到的是安心而不是满意。如果最终得到的是粗粒度的控制器方法，那么模拟 HTTP 上下文就会成为一个持续不断的需求。它不仅是 ASP.NET MVC 框架所需要的，它最终也会是你自己所需要的。

然而，话虽如此，有时候你却真的需要模拟 HTTP 上下文。让我们看看怎么进行。

1. 模拟 HttpContext 对象

HttpContext 对象继承自 HttpContextBase，你所需要做的就是为它创建一个模拟。HttpContext 对象是对象引用的一个普通聚合；它几乎不需要包含额外的代码。所以，大多数情况下模拟都会运行良好。下面是如何使用 Moq 构建一个伪造 HTTP 上下文的代码：

```
public void BuildHttpContextForController(Controller controller)
{
   var contextBase = new Mock<HttpContextBase>();
   var request = new Mock<HttpRequestBase>();
   var response = new Mock<HttpResponseBase>();
   var server = new Mock<HttpServerUtilityBase>();
   ...

   contextBase.Setup(c => c.Request).Returns(request);
   contextBase.Setup(c => c.Response).Returns(response);
   contextBase.Setup(c => c.Server).Returns(server);

   // Pass the fake context down to the controller instance
   var context = new ControllerContext(
      new RequestContext(contextBase.Object, new RouteData()), controller);
   controller.ControllerContext = context;
```

```
    return;
}
```

这只是模拟 ASP.NET 内部对象的第一层而已。例如，每当应用程序在单元测试中查询 Request 时，它就会获得模拟对象。但是模拟对象是一个虚拟对象，并且需要其自己的设置。让我们来看几个例子。

2. 模拟 Request 对象

你可能希望按照与一些特定成员有关的预期来扩展模拟的 Request 对象。例如，下面是如何在测试中模拟 GET 或 POST 请求的代码：

```
var method = "get";
contextBase.Setup(c => Request.HttpMethod).Return(method);
```

在本章前面讨论测试路由时，我们也碰见过类似的代码：

```
var url = ...;
contextBase.Expect(c => Request.AppRelativeCurrentExecutionFilePath).
                   Return(url);
```

你大概不想使用 Request.Form 对象从控制器内部读取提交的数据，因为你可能会发现模型绑定器更有效。然而，如果一个控制器方法中有一个对 Request.Form[“MyParam”]的旧式调用，你该如何测试它呢？

```
// Prepare the fake Form collection
var formCollection = new NameValueCollection();
formCollection["MyParam"] = ...;

// Fake the HTTP context and bind Request.Form to the fake collection
var contextBase = new Mock<HttpContextBase>();
contextBase.Setup(c => c.Request.Form).Returns(formCollection);

// Assert
...
```

这样，每一次你的代码通过 Request.Form 读取内容，它实际最终都是从为测试目的而提供的名称/值集合中读取的。

3. 模拟 Response 对象

让我们来看几个与 Response 对象有关的例子。例如，你可能想通过将 HttpResponse 对象强制写入文本编写器对象来模拟 Response.Write 调用，如下所示：

```
var writer = new StringWriter();
var contextBase = new Mock<HttpContextBase>();
contextBase.Setup(c => c.Response).Return(new FakeResponse(writer));
```

在此种情况下，FakeResponse 类的使用如下所示：

```
public class FakeResponse : HttpResponseBase
{
    private readonly TextWriter _writer;
    public FakeResponse(TextWriter writer)
    {
        _writer = writer;
    }

    public override void Write(string msg)
    {
        _writer.Write(msg);
    }
}
```

有了这段代码，你就可以对具有 Response.Write 调用的控制器方法进行测试了，如下面所示的这个：

```
public ActionResult Output()
{
    HttpContext.Response.Write("Hello");
    return View();
}
```

以下是其测试：

```
[TestMethod]
public void Should_Response_Write()
{
    // Arrange
    var writer = new StringWriter();
    var contextBase = new Mock<HttpContextBase>();
    contextBase.Setup(c => c.Response).Returns(new FakeResponse(writer));
    var controller = new HomeController();
    controller.ControllerContext = new ControllerContext(
                contextBase.Object, new RouteData(), controller);

    // Act
    var result = controller.Output() as ViewResult;
    if (result == null)
```

```
        Assert.Fail("Result is null");

    // Assert
    Assert.AreEqual("Hello", writer.ToString());
}
```

类似的，如果要让某些属性或方法返回特定的值，就可以配置一个动态生成的模拟。下面是两个例子：

```
var contextBase = new Mock<HttpContextBase>();

// Mock up the Output property
contextBase.Setup(c => Response.Output).Returns(new StringWriter());

// Mock up the Content type of the response
contextBase.Setup(c => Response.ContentType).Returns("application/json");
```

另一方面，对于 cookies，你可能希望同时在 Request 和 Response 上模拟 Cookies 集合以返回 HttpCookieCollection 类的一个新实例，它将充当你单元测试范围的 cookie 容器。在本章后面讨论如何用操作筛选器测试控制器方法时，我会介绍更多有关模拟 Response 对象的信息。

4. 模拟 Session 对象

模拟更易于使用，但有时你需要给模拟对象的各种方法分配一个行为。当行为如返回一个指定值这样简单时，这是容易做到的。但是，为了有效测试该方法是否正确地更新会话状态，你需要提供一个内存中的对象来模拟原始对象的行为和具备存储信息的能力，而这却不是一个简单的模拟任务。不过，使用一个伪造会话类便可以使其简单明了。下面是用于会话状态的一个简约却有效的伪造：

```
public class FakeSession : HttpSessionStateBase
{
    private readonly Dictionary<String, Object> _sessionItems =
                                   new Dictionary<String, Object>();

    public override void Add(String name, Object value)
    {
        _sessionItems.Add(name, value);
    }

    public override Object this[String name]
    {
        get { return _sessionItems.ContainsKey(name) ? _sessionItems[name] :
```

```
            null; }
        set { _sessionItems[name] = value; }
    }
}
```

下面的代码显示了如何安排一个测试：

```
[TestMethod]
public void Should_Write_To_Session_State()
{
    // Arrange
    var contextBase = new Mock<HttpContextBase>();
    contextBase.Setup(c => c.Session).Returns(new FakeSession());
    var controller = new HomeController();
    controller.ControllerContext = new ControllerContext(
                    contextBase.Object, new RouteData(), controller);

    // Act
    var expectedResult = "green";
    controller.SetColor();   // Runs Session["PreferredColor"] = "green";

    // Assert
    var result = controller.HttpContext.Session["PreferredColor"];
    Assert.AreEqual(result, expectedResult);
}
```

如果控制器方法仅读取自 Session，那么你的测试可以更简单，可以避免完全伪造 Session。下面是一个控制器操作示例：

```
public ActionResult GetColor()
{
    var o = Session["PreferredColor"];
    if (o == null)
        ViewData["Color"] = "No preferred color";
    else
        ViewData["Color"] = o as String;

    return View("Color");
}
```

下面的代码片段显示了测试刚才所示方法的一种可能方式：

```
// Arrange
var contextBase = MockRepository.GenerateMock<HttpContextBase>();
contextBase.Expect(s => s.Session["PreferredColor"]).Return("Blue");
```

```
var controller = new HomeController();
controller.ControllerContext = new ControllerContext(
                              contextBase.Object, new RouteData(), controller);

// Act
var result = controller.GetColor() as ViewResult;
if (result == null)
   Assert.Fail("Result is null");

// Assert
Assert.AreEqual(result.ViewData["Color"].ToString(), "Blue");
```

在这种情况下，可以指示HTTP上下文模拟在请求其Session属性提供用于“PreferredColor”条目的值时返回“Blue”字符串。

对于控制器的方法需要读取和写入会话状态这一更常见的方案，你需要使用一些基于FakeSession类的测试解决方案。

5. 模拟 Cache 对象

模拟 ASP.NET Cache 对象是一个值得注意的任务，虽然模拟缓存层不需要新的方式。HttpContextBase 类有一个 Cache 属性，但你不能模拟它，因为该属性并不代表 ASP.NET 缓存系统的一个抽象；相反，它是特定类的具体实现。下面的代码显示了 Cache 属性是如何在 HttpContextBase 类上进行声明的：

```
public abstract class HttpContextBase : IServiceProvider
{
    public virtual Cache Cache { get; }
    ...
}
```

Cache 属性的类型实际上是 System.Web.Caching.Cache——真正的缓存对象，而不是一个抽象。更不幸的是，Cache 类型是密封类，因此不具备可模拟性，也无法在单元测试中使用。

为此你能做什么呢？有两个选项。一个选项需要使用能够处理密封类的测试工具。举例来说，其中的一个工具是 Typemock Isolator(一款商业产品)；另一个工具是微软 Moles。另一种可选项是使用一个包装类从你打算测试的任何代码中执行对 Cache 的访问。第 5 章检验了这种方法。

根据你的所见，创建一个实现指定接口的缓存服务对象；比如 ICacheService。接着，你要在 global.asax 中向应用程序注册该类，并添加一个公共的静态属性来读写缓存：

```
protected void Application_Start()
{
```

```
    ...

    // Inject a global caching service(for example, one based on ASP.NET Cache)
    RegisterCacheService(new AspNetCacheService());
}

private static ICacheService _internalCacheObject;
public void RegisterCacheService(ICacheService cacheService)
{
    _internalCacheObject = cacheService;
}

public static ICacheService CacheService
{
    get { return _internalCacheObject;  }
}
```

在控制器方法中，你要完全停止使用 HttpContext.Cache。而你的控制器会有以下设计：

```
public partial class HomeController : Controller
{
    public ActionResult SetCache()
    {
       // MvcApplication is the name of the global.asax class; change it at will
       MvcApplication.CacheService["PreferredColor"] = "Blue";
       return View();
    }
    ...
 }
```

如何对此进行测试呢？下面有一个示例：

```
[TestMethod]
public void Should_Write_To_Cache()
{
    // Arrange
    var fakeCache = new FakeCache();
    MvcApplication.RegisterCacheService(fakeCache);

    // Act
    controller.SetCache();

    // Assert
    Assert.AreEqual("Blue", fakeCache["PreferredColor"].ToString());
}
```

FakeCache 类可以是这样的：

```
public class FakeCache : ICacheService
{
    private readonly Dictionary<String, Object> _cacheItems =
                new Dictionary<String, Object>();

    public object this[String name]
    {
        get
        {
            if (_cacheItems.ContainsKey(name))
                return _cacheItems[name];
            else
                return null;
        }
        set { _cacheItems[name] = value; }
    }
}
```

用这种方式，你就可以测试控制器方法和服务，甚至当缓存是分布式缓存时也可以使用缓存数据。缓存服务隐藏了所有细节，并且使应用程序更具有扩展性和可测试性。

9.4　本章小结

可测试性是软件的一个基本方面，正如 ISO/IEC 9126 论文在 1991 年所认可的那样。在 ASP.NET MVC 中，对代码的可测试性设计更为容易且更受支持。但是，也可以在 ASP.NET Web Forms 中编写测试代码并在很大程度上进行测试。可测试性是追求优秀设计的极佳理由。设计就这样产生了差别。

我们都赞同编写测试是有益的、值得推荐的、甚至是富有魅力的。然而，除非你将其与一些测试驱动的设计方法相关联，否则编写测试本身就会面临耗费极大工作量的风险。一方面，你有要测试的类；另一方面，你还有要确保能够测试其他类的类。并且，你没有任何中间层可以保证测试是适当和有意义的。

创建用于更有效重构的环境同时不排除单元测试的方式是基于软件协议。软件协议会为每个类中的每个方法定义条款和条件。这就为你提供了一个可选的运行时检查机制，以及在必要时如何重构的具体指导。软件协议在软件中并不是一个新概念，但它们是在.NET 4 中才开始得到广泛实施的。请在 MSDN 杂志的 Cutting Edge 专栏查看代码协议的相关文档和我写的有关这一主题的文章，其网址是 http://msdn.microsoft.com/en-us/magazine/default.aspx。

第 10 章

Web API 的执行指南

空想者欺骗自己，而骗子只欺骗其他人。

——Friedrich Nietzsche

ASP.NET Web API 是一个新的框架，其明确意图是支持和简化可以由各种客户端所使用的 HTTP 服务的构建过程；尤其是 HTML 网页和移动应用程序。Web API 背后的核心理念并不新鲜，开发人员和架构师面对同样的挑战至少已经十年了。一路走来，已经制定了几种框架和技术，每一种都常常被作为这一方面的最终解决方案。你多少次听说过 Windows 通信基础(WCF)是为各种类型的客户端打造包括 TCP 和 HTTP 在内的服务层的理想技术了？

在这方面，Web API 至少是最新的尝试——并且希望会是最后的真正彻底的尝试——为经由 HTTP 的 Web 服务提供理想的框架。

Web API 的应用范围非常广泛，对任何开发人员来说都是令人信服和有用的，不仅仅是 Web 开发人员。但是，出于一些原因，Web API 有时候会被当作与 ASP.NET MVC 相关的工具被介绍，或者让人感觉是与 ASP.NET MVC 相关的工具。这未免有些误导；加之，如果看看核心功能，ASP.NET MVC 开发人员是微软市场领域中唯一能在没有 Web API 的情况下继续愉快开发的人了。

Web API 是一个精心设计的框架，它用于为.NET 应用程序构建 RESTful(具象状态传输)和远程过程调用(RPC)风格 HTTP 服务。Web API 贯穿了 ASP.NET MVC 的各个方面，比如路由、安全、控制器和可扩展性，还有一些并不受普通 ASP.NET MVC 直接支持的特殊领域。总之，Web API 应该有一本恰如其分地叙述其所涉及内容的书。这一章会阐述 Web API 关键因素的执行概要——如何构建和使用 HTTP 服务——从 ASP.NET MVC 开发人员的角度。更多示例以及内部架构的更深层次内容，我建议你看看 http://www.asp.net/web-api 的优秀文档。

10.1 Web API 的来龙去脉

Web API 不是一夜之间出现的框架，当然更不是根据什么神灵启示而创造出来的框架。

简单地说，它有希望成为大约十年前就已经开始的演变进程的最后一站，其目标是提供标准化的 Web 服务。

10.1.1　标准化 HTTP API 的需求

在被绑定到特定的技术和框架之前，Web 服务一词仅仅用于描述在 Web 上可用的软件服务。共享返回数据的而非标记数据的 HTTP 端点的需求在 20 世纪 90 年代后期就已经非常强烈和明确了，当时 ASP.NET 框架正在制定中。

就像常常发生在软件开发中的那样，架构师设法将特定的需求放入越来越广泛的上下文中，最终创建出超越了解决最初需求的技术和模式。经过几年的发展后，软件业界又回到了最初的需求——“我们需要的是最简单的基础架构，可以允许我们通过 HTTP 调用端点。”令人惊讶的是，在 ASP.NET 和微软堆栈内部以及周边还没有这种基础架构。相反，有大量的技术可以让开发人员用作 HTTP 服务来操作，每种技术都有其自身的优缺点。

Web API 解决了这一问题，它为开发人员提供了一个有效而独特的平台来通过 HTTP 公开应用程序服务。“应用程序服务”的样式和结构的定义取决于架构师。它可以是将应用程序发布到 Web 的软件开发工具包(SDK)。或者，也可以是 Web 上的应用程序的 SDK，可以用可控的方式访问，但又不公开可用。最后，你还可以使用 Web API 发布一个面向服务架构(SOA)形式的服务——一个没有用户界面的自主、独立的应用程序，只以各种格式服务于数据。

1. 超越 WCF

当整个世界认识到 Web 服务的能力时，一种标准协议就很快产生了，该协议成为了 Web 服务可以与调用方进行交换的媒介：简单对象访问协议(SOAP)。最重要的是，很多更深入的规范成为了工作进程的标准，统称为 WS-*协议。

WCF 的最初构想是透过种类繁多的传输层来支持 SOAP 和 WS-*，这些传输层包括 TCP、消息队列(MSMQ)、命名管道，以及最后也是最重要的一个：HTTP。

不难看出，虽然 WCF 有自己坚实的发展动机和理由，但开发人员大多只是将它用作 HTTP 端点的快捷方式而已。这便对 WCF 基础架构提出了更有效支持非 SOAP 服务的要求，以达到能够通过 HTTP 来服务纯 XML、文本和 JavaScript 对象标记(JSON)的目的。这些年来，我们先是从 WCF 团队获得了 webHttpBinding 绑定机制，接着又出现了贴附式框架，比如 REST 入门套件。

归根结底，这是提供一些语法技巧的问题，以帮助将 HTTP 用作单纯运输层这一想法更容易付诸实践。WCF 之上的 HTTP 设施并没有消除开发人员所面临的障碍，比如声名狼藉的 WCF 过度配置、对特性的过度使用以及其结构设计未特别考虑到可测试性方面。

主要的变化是将多传输服务从普通的 HTTP 服务中分离出来，并且消除了 WCF 的所有厚重机制，创建了一个精简且专注于 HTTP 框架的 HTTP 服务。这就是 Web API。

注意：

随着服务重点从 WCF 转移，是否意味着 WCF 就变成了丧失生命力的技术，没有实际的应用程序会继续使用了呢？当然不是。当你切实需要公开一个可以通过协议调用而非 HTTP 调用的服务时，WCF 就是唯一重要的选择。当你真正需要如安全和事务处理支持这样的高级功能时， WCF 也仍然是一个重要的解决方案。

2. Web API 对客户端应用程序的意义

Web API 非常适合于现在似乎很常见的应用程序场景：客户端应用程序需要调用远程后端来下载数据或请求处理。客户端应用程序可以采用多种形式： JavaScript 密集型网页、富客户端或移动应用程序。

采用这些形式，HTTP 协议几乎可以和 SOAP 一样有效，却更为简单和快速。可以使用 HTTP 动词和标头来标识要对远程资源所执行的操作；如果喜欢专注于操作的方式，可以使用 URL 模板来表达语义。无论哪种方式，你都要使用消息主体来序列化所有的输入值和接收内容。JSON 格式是在客户端和 HTTP 服务之间序列化对象的理想格式。

3. Web API 对 ASP.NET Web Forms 应用程序的意义

无论喜欢与否，使用 ASP.NET Web Forms 的网站数量仍远远大于基于 ASP.NET MVC 的网站数量。在 ASP.NET 4 及更新的版本中，微软设法简化了问题，比如加入了生成 HTML 的受限访问、对正在生成的 HTML 进行更好的控制、减轻视图状态和路由的影响等。不过，并未得到解决的一个问题是，如何让 Web Forms 开发人员能够快速公开 HTTP 可调用端点。

这并不是说在 Web Forms 中不能公开 HTTP 端点以从客户端页面回调；可以将 HTTP 处理程序或 WCF 服务用于自由生成 JavaScript 代理类。归根结底，它是可以达成该任务的，但其编码过程让开发人员越来越感觉到烦恼。

Web API 使得从 Web Forms 应用程序中公开 HTTP 服务变得像在 ASP.NET MVC 中使用控制器一样简单：只要添加一个类并让它根据预定义的路由规则运行即可。不喜欢默认的路由规则？没关系，添加你自己的规则也很简单！

Web API 为 Web Forms 开发人员带来了巨大的好处。不过同时，它也代表了 ASP.NET MVC 开发人员的困惑来源。实际上，常常碰到的问题是，我为什么应该使用它呢？相较于普通的控制器，Web API 能带来什么好处？

10.1.2　MVC 控制器与 Web API 对比

一个 ASP.NET MVC 应用程序产生自多个控制器类的组合。通常，每个控制器类会公开多个操作以便用户界面可以通过 URL 调用。请求处理与响应生成之间的有序分离使得控制器类以各种格式返回响应成为了可能，这些格式包括 HTML、JSON、XML 以及普通文本。

针对此，在 ASP.NET MVC 应用程序中使用 Web API 有什么意义呢？

1. 控制器类完成所有工作

如果想要从 ASP.NET MVC 应用程序内部返回一些 JSON 数据，你只需要在新的或已有的控制器类中创建一个特设方法即可。对于这个新方法的唯一特殊要求是返回一个 JsonResult 对象，如下所示：

```
public JsonResult LatestNews(int count)
{
    var listOfNews = _service.GetRecentNews(count);
    return Json(listOfNews, JsonRequestBehavior.AllowGet);
}
```

该方法也可以是父 ActionResult 类，除了声明该方法的返回类型之外，更重要的是对 Json 方法的调用。Json 方法会确保指定对象是在 JsonResult 对象中封装的。在其从控制器类返回之后，JsonResult 对象就会被负责当前请求的操作调用程序处理。

整体作用是，被包含的对象——由控制器最先计算的——被序列化到 JSON，并且嵌入到发送回请求设备的响应主体中。同样，可以为普通文本、二进制内容，或者以你喜欢的方式格式化的 XML 内容提供服务。

2. 了解不同之处

在 ASP.NET MVC 中，你仅会因为要将新控制器类或特定方法添加到已有控制器类而使用 HTTP 服务。这从第 1 个版本开始就是有效的并且到目前最新版本的 ASP.NET MVC 中仍然有效。那么你为什么还应该看看其他东西呢？

Web API 框架依赖于一个不同的运行时环境，该环境完全是从 ASP.NET MVC 的运行时环境中分离出来的。这样做的目的就是为了让非 ASP.NET MVC 应用程序可以使用 Web API。该运行时环境必然很大程度上受到了 ASP.NET MVC 的启发，但总体来看更加简单并且更为直截了当，因为它被预期仅提供服务而非标记。

从 ASP.NET 开发人员的视角来看，以下三个要点总结了 Web API 方式相对于 ASP.NET MVC 的好处：

- **从结果序列化中解耦代码**　这指的是 Web API 控制器需要你从每个方法中仅返回数据而不在方法内部对结果进行序列化处理的情况。
- **内容协商**　称为格式化器的一套新系列组件会负责序列化返回到请求设备的数据。更为相关的是，格式化器是根据传入请求 Accept 标头的内容自动选择的。提供了内置的 XML 和 JSON 格式化器；替换它们是一项简单的配置任务。该特性简化了可能会以各种格式返回相同原始数据的方法的开发，最典型的是 XML 和 JSON 格式。

- **在互联网信息服务(IIS)之外托管**　从某种程度上讲，内容协商这一特性也能在普通的 ASP.NET MVC 中实现。但是，如果选择在 ASP.NET MVC 应用程序中实现你的 HTTP 服务，那么你就要绑定到 IIS 作为 Web 服务器；换句话说，你不能在其他地方托管你的 API。这是由于 ASP.NET MVC 原生就是绑定到 IIS 的，因为它最初旨在作为框架为 Web 应用程序提供服务。但是，Web API 并不依赖 IIS，并且可以在你自己的托管进程中对其进行自托管，比如 Windows 服务或控制台应用程序(这一特性使得 Web API 服务接近于 WCF 服务)。

从本质上讲，Web API 并没有真的为 ASP.NET MVC 开发带来太多潜力，所以也不要将其看作是一个必要特性，例如，它在 Web Forms 开发中就并非必要特性。通常，它的使用依赖于特定项目的需求，并且一定程度上取决于开发团队的偏好。

3. 构建 RESTful 应用程序

如果过去曾努力使 ASP.NET MVC 控制器尽可能贴近 RESTful，你会发现通过选择 Web API 并相应地组织 Web 应用程序会使该任务更易达成。例如，可以使用主要是客户端的解决方案(比如单页面应用程序)或者许多服务器端网页来通过该 API 的直接 HTTP 调用完成大多数工作，以创建、读取、更新和删除(CRUD)数据。

要构建一个 RESTful 应用程序，你必须识别出应用程序中的不同资源并将基于这些资源执行的操作映射到 HTTP 方法进行处理。有了 Web API，这在很大程度上就变成自然而然的了；你所要做的就是，定义一个数据转换对象以返回到客户端并制定一个在宿主 ASP.NET MVC 应用程序中的类以基于该对象执行基础的 CRUD 操作。Web API 能保障解决方案的 RESTfulness，它确保了将 HTTP PUT 请求映射到创建操作、将 HTTP GET 请求映射到读取操作、将 HTTP POST 请求映射到更新操作，以及将 HTTP DELETE 请求映射到删除操作。

言归正传，让我们继续进行一些示例代码的介绍。

10.2　让 Web API 开始工作

从微软 Visual Studio 中，首先你要创建一个 Web API 类型的 ASP.NET MVC 项目。你会得到一个具有两个主要特征的 ASP.NET MVC 应用程序。第一，App_Start 项目文件夹包含一个用于 Web API 以设置特设路由的初始化程序组件。第二，Controllers 文件夹包含两个示例控制器类：普通的 HomeController 类以及特定于 Web API 的 ValuesController 类。后者继承自 ApiController 而非 Controller。

这是 Web API 的支柱。Web API 模块是控制器类的一个集合，这些控制器类继承自预定义的 ApiController 而不是 Controller。图 10-1 揭示出，Web API 基础架构与 ASP.NET MVC 基础架构是独立的，不具有隐藏的依赖性。

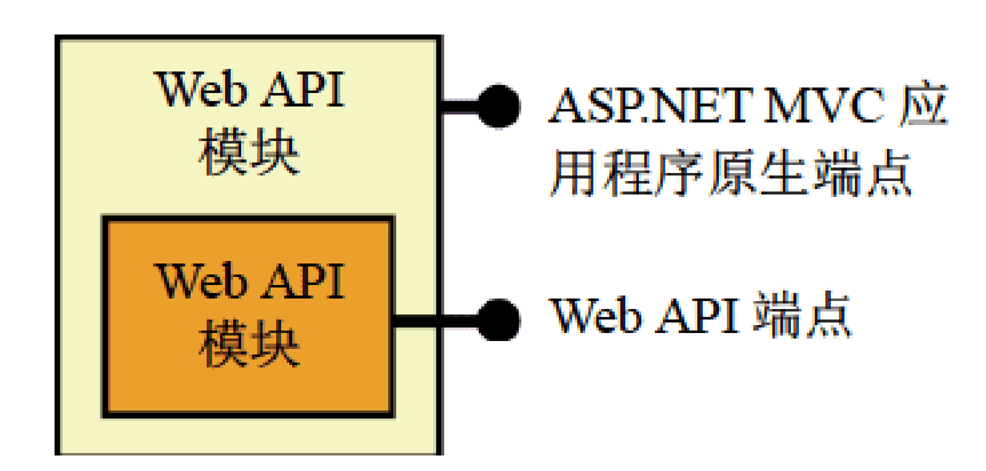


图 10-1　Web API 可通过不同的应用程序来托管并且与 ASP.NET 并非紧密耦合

Web API 模块是一个独立的基于 HTTP 的服务，能够被许多应用程序托管，而 ASP.NET MVC 应用程序大概是最常用的宿主类型。一个关键要点是：不要被几乎到处出现的“controller”一词所蒙蔽。归根结底，嵌入由 Web API 驱动的 HTTP 端点集的 ASP.NET MVC 应用程序使用了普通的 Controller 类以派生基于 HTML 的控制器，并且使用了 ApiController 以派生返回要格式化成 JSON 或 XML 的普通数据的控制器类。让我们看看如何构建一个宿主 ASP.NET MVC 应用程序中的 Web API 模块。

10.2.1　设计 RESTful 接口

Web API 使用了大量的约定。根据默认约定，产生的编程风格本质上就是 RESTful 风格。这表明了与普通 ASP.NET MVC 控制器对应的另一个不同之处，普通 ASP.NET MVC 控制器大多数都是面向 RPC 的。确切地说，Web API 完全不会限定于 RESTful 风格；在本章稍后你将看到，可以选择性地赋予 Web API RESTful 风格或 RPC 风格。但是，默认的并且最受追捧的风格基本就是 RESTful。

1. 定义资源类型

REST 纯粹主义者主张(并且有合理的理由)，CRUD 甚至都不是 REST 定义的一部分。REST 被定义为专注于通过 Web 协议、特别是 HTTP 协议识别和处理资源的一种架构风格。抽象理论和思想体系就讲这么多了；但面临构建遵循良好 REST 原则的软件标准实现时，你真的就是要构建基于 HTTP 的 CRUD。

无论如何，REST 完全在于给出资源的表示并且在要应用到其他资源时将服务器操作关联到通用的 HTTP 动词。

在 Web API 的上下文中，这仅仅意味着 Web API 模块是一个由许多控制器类组成的类库。这些控制器类继承自 ApiController，ApiController 是一个在新的 System.Web.Http 程序集中定义的类，它并非继承自 ASP.NET MVC Controller 类。当使用默认 RESTful 风格进行构建时，Web API 控制器类会公开一个编程接口，该接口要提供对指定资源类型的 CRUD。最有可能的情况是，你最终会为倾向于编程方式处理的每个资源类型使用一个控制器类。

资源类型通常被定义为一个普通数据传输对象(DTO)，本质上是一个普通 C#类。Visual Studio 向导为你生成的示例 Web API 控制器类最初不具有一个真实的资源类型。示例控制器仅会使用普通字符串形式的值。

```
public class ValuesController : ApiController
{
    public IEnumerable<string> Get()
    {
       return new [] { "value1", "value2" };
    }

    public string Get(int id)
    {
       return "value";
    }
    ...
}
```

在更实际的例子中，你可能会想要使用如下定义的一些 News 类：

```
public class News
{
   public String Title { get; set; }
   public String Content { get; set; }
   public DateTime Published { get; set; }
}
```

接着，可以使用一个处理 News 实体的控制器，比如以下代码所示的一个控制器：

```
public class NewsController : ApiController
{
    public IEnumerable<News> GetAll()
    {
       var url = ...;
       var client = new WebClient();
       var rss = client.DownloadString(url);
       var news = ParseRssInternal(rss);
       return news;
    }

    public News Get(int index)
    {
       var all = GetAll();
       return all[index];
    }
    ...
}
```

归根结底，Web API 控制器类是公共方法的一个集合。当不是原始数据的时候，每个方

法仅会接受和返回 DTO。像 NewsController 这样的普通类如何才能具有通过 URL 调用并返回像序列化成 JSON 或 XML 这样的值的方法呢？这一切都是由于你拥有运行时 Web API 管道(例如路由)并且该任务是由 ApiController 类在后台完成才得以实现。

2. ApiController 类

Web API 控制器不返回视图；相反，他们会向调用方返回数据。调用方就是连接到 HTTP 上端点的任意客户端；例如，一个网页、C#类或移动应用。返回值就是 Web API 控制器与 ASP.NET MVC 控制器之间最明显的差异。Web API 中的控制器基类结合了 ASP.NET MVC 中操作调用程序和控制器的行为。

第 1 章“ASP.NET MVC 控制器”揭示出，由 ASP.NET MVC 运行时所捕获的请求会被处理成由控制器名称和操作名称指定的一对值。操作调用程序是负责获取控制器类新实例的系统组件，它会调用方法并在结果之上执行预期操作。在结果之上执行操作实质上指的是为调用方准备响应。通常，这涉及生成 HTML 标记、序列化到 JSON 或配置 HttpResponse ASP.NET 对象用于重定向。

在 Web API 中，所有这些任务都是在 ApiController 类内部通过 ExecuteAsync 方法进行协调的。此外，这还意味着在 Web API 中每个请求都是异步处理的。不过请注意，这并非完全表示每个由请求所触发潜在长期运行的任务都是异步的。

ApiController 类还能访问 Web API 环境的当前配置并且恰如其分地使用该信息。配置容器是新 HttpConfiguration 类的一个实例，它会存储注册的用于 XML 和 JSON 的格式化器、路由、绑定规则和筛选器的列表。

注意：

Web API 基础架构，与最值得注意的 ApiController 类及相关类，都在新的 System.Web.Http 程序集中托管。正如所介绍的，此程序集不依赖于 System.Web 和 System.Web.Mvc。任何听起来类似于 ASP.NET MVC 的概念(例如，操作筛选器、路由、绑定)都是为 Web API 的目的而重新实现的。

3. 路由到操作方法

当 Web API 框架接收到 HTTP 请求时，它会根据操作方法将其解析到控制器类上的调用。为确定要调用的操作，类似于经典 ASP.NET MVC，Web API 框架会使用一个路由表。默认项目模板中的 WebApiConfig 类包含了以下默认路由：

```
public static void Register(HttpConfiguration config)
{
   config.Routes.MapHttpRoute(
      name: "DefaultApi",
```

```
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
}
```

如你所见，由 Web API 识别的 URL 的默认格式包含了紧随服务器名称之后的 api 令牌以及控制器名称。根据前面提及的 NewsController 类，一个有效的 URL 可能是：

```
http://server/api/news/1
```

如果具备坚实的 ASP.NET MVC 背景知识，你就应该马上会注意到一个丢失的元素：操作名称。这就是 Web API 与 ASP.NET MVC 之间的另一个明显不同。默认情况下，Web API 框架会使用 HTTP 方法来选择操作而非 URL。这就是让整个框架 RESTful 化方面的内容。

当请求上面的 URL 时到底会在 NewsController 上调用哪个方法呢？

要找出该方法，Web API 框架会查找与请求的 HTTP 方法上的匹配。如果通过 HTTP GET 方法进行请求，则 Web API 会查找一个名称以 Get 开头的操作。如前所示，在 NewsController 类中，有两个名称以 Get 开头的方法：GetAll 与 Get。要选取哪一个呢？这取决于与第 3 章“模型绑定架构”中讨论的 ASP.NET MVC 绑定规则类似的绑定规则。

根据默认路由，这两个方法在名称方面都通过了测试。然而，由于默认路由，方法可以具有一个可选的名称为 id 的尾随参数。GetAll 方法没有参数，因而代表着一个良好匹配。相反，Get 方法需要一个名为 index 的参数。由于这个参数不能在 URL 中找到，所以该方法不能匹配。

4. 处理多个匹配

值得注意的是，方法签名中小小的变化就能避免图 10-2 中描绘的异常。例如，将 Get 方法的 index 参数转变成一个带有默认值的参数，如下所示：

```
public News Get(int index=1)
{
   ...
}
```

然后，以下 URL 就能同时由 Get 与 GetAll 匹配：

```
http://server/api/news/1
```

当出现这种情况时，Web API 会在内部抛出一个异常，并且由调用方接收的状态代码是图 10-2 中所示的带有 JSON 错误消息的 HTTP 500，该消息嵌入在响应主体中。

```
▼<Error>
   <Message>An error has occurred.</Message>
  ▼<ExceptionMessage>
     Multiple actions were found that match the request: Simplest.Models.Dto.News Get(Int32) on type Simplest.Api.NewsController
     System.Collections.Generic.IList`1[Simplest.Models.Dto.News] GetAll() on type Simplest.Api.NewsController
   </ExceptionMessage>
   <ExceptionType>System.InvalidOperationException</ExceptionType>
  ▼<StackTrace>
     at System.Web.Http.Controllers.ApiControllerActionSelector.ActionSelectorCacheItem.SelectAction(HttpControllerContext controllerContext) at
     System.Web.Http.Controllers.ApiControllerActionSelector.SelectAction(HttpControllerContext controllerContext) at
     System.Web.Http.ApiController.ExecuteAsync(HttpControllerContext controllerContext, CancellationToken cancellationToken) at
     System.Web.Http.Dispatcher.HttpControllerDispatcher.SendAsyncInternal(HttpRequestMessage request, CancellationToken cancellationToken) at
     System.Web.Http.Dispatcher.HttpControllerDispatcher.SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)
   </StackTrace>
 </Error>
```

图 10-2　在多个匹配的情况下返回的 JSON 错误消息

对于操作名称完全不匹配的情况，碰到多个操作方法匹配请求的几率是很高的。如果将 index 参数重命名为 id，就像路由中的一样，则也会抛出多个操作的异常。在这种情况下，由于 id 是可选的，所以名称以 Get 开头的带有单个参数的任何方法都会成为服务于请求的候选项。

5. 主动遵循命名约定

所以，看起来你在为操作方法寻找合适名称方面就会很辛苦。这恰恰是问题的关键所在，并且该问题来自于默认约定。例如，如果是通过使用 HTTP POST 动词进行请求的，那么为了让请求得以成功处理，就应该正好只有一个名称以 Post 开头的方法。同样的默认约定还需要应用到来自 GET、PUT 与 DELETE 动词的请求。

注意：

要避免一个方法被作为操作调用，你要使用 NonAction 特性。这将指示 Web API 框架，即便该方法是公共的并且匹配路由规则，也不应该将其用作操作方法。

10.2.2　预期的方法行为

Web API 框架构建了一些与每个方法行为有关的会匹配 HTTP 动词的预期内容。正如你已经看到的，对于一个 HTTP GET 方法，所需要的就是该方法返回一些序列化.NET 对象。对于其他常见的 HTTP 动词，比如 PUT、POST 以及 DELETE，就有一点复杂了。

1. POST 方法的语义

POST 方法通常预期要将新资源添加到一些后台存储。其返回类型是 HttpResponseMessage，它表示一个 HTTP 响应消息并同时包括要返回给调用方的状态代码以及实际内容。下面是 POST 方法常见的签名和实现：

```
public HttpResponseMessage PostNews(News news)
{
   // Do something here to store the news
   var newsId = SaveNewsInSomeWay(news);
```

```
    // Build an empty response: 201 is code for Created
    var response = Request.CreateResponse<String>(HttpStatusCode.Created,
       "OK");

    // Store location of the new item
    var relativePath = String.Format("/api/news/{0}", newsId);
    response.Headers.Location = new Uri(Request.RequestUri, relativePath);
    return response;
}
```

一开始，该方法应该执行正确插入或保存任何接收到的数据的任务。接着，它应该设法创建一个响应对象。该状态代码应该被设置为 Created(如果实际创建了新资源的话)并且该内容应该是要为调用方返回的消息。还建议将新创建的项添加到响应标头集合这一位置。

2. PUT 方法的语义

PUT 方法通常预期要更新某后台存储中的一个现有资源。如果不想要对调用方共享反馈信息的话，其返回类型可以为空。不过，合理地说，你应该返回被设置为 HTTP 200 或 HTTP 204 的 HttpResponseMessage 实例，就如以下代码一样：

```
public HttpResponseMessage PutNews(Int32 id, News news)
{
    if (id <= 0)
       throw new HttpResponseException(new HttpResponseMessage
                                      (HttpStatusCode.NotFound));

    // Do something here to update the news: consider returning HTTP 200
    var response = Request.CreateResponse<String>(HttpStatusCode.OK, "OK");
    return response;
}
```

返回 HTTP 204 这个状态代码也是可以接受的，该状态代码表明，请求已经成功处理但有意将响应设置为空。要选择的 HttpStatusCode 值是 NoContent。

3. DELETE 方法的语义

DELETE 方法通常预期要删除某后台存储中的一个现有资源。请记住虚方法是可行的，但正如以下代码中所示的，你的目标应该是返回 HTTP 200 或 HTTP 204。在这种情况下，HTTP 200 或 204 将被解读为资源已经成功删除的证明。

```
public HttpResponseMessage DeleteNews(Int32 id)
{
```

```
    if (id <= 0)
        throw new HttpResponseException(new HttpResponseMessage
                                            (HttpStatusCode.NotFound));

    // Do something here to delete the news

    return new HttpResponseMessage(HttpStatusCode.NoContent);   // 204
}
```

对于一个 DELETE 请求，你还可以选择返回状态代码 HTTP 202，它表明请求已被服务器接受并且资源已被标记为删除。重要的一点是，这不能让调用方确定资源是否被真正移除，但同时它能够避免服务器承担删除的职责。

4. 其他方法的语义

尽管 GET、PUT、POST 与 DELETE 是最常用的 HTTP 动词，但可行的 HTTP 动词列表并非只包含这四个。不过，从 Web API 的角度讲，除了以上 4 个 HTTP 动词之外，其他合理的动词并不会被框架特别处理。这意味着没有约定能够应用于它们。看看如下代码：

```
public HttpResponseMessage HeadNews(int id)
{
    var message = new HttpResponseMessage(HttpStatusCode.OK);
    message.Headers.Add("NewsId", id.ToString());
    return message;
}
```

根据万维网联盟(W3C)的表述，HEAD 请求预期要像 GET 那样运行，不同之处在于其响应主体为空并且只会返回响应标头。

但是，前面的代码并不会运行且会产生一个 HTTP 404。原因是我们讨论的适用于最常用动词的命名约定并不能应用到此处。如果想要该方法为传入 HTTP HEAD 的请求运行，应该使用 AcceptVerbs 特性。

```
[AcceptVerbs("HEAD")]
public HttpResponseMessage HeadNews(int id)
{
   ...
}
```

再次注意，AcceptVerbs 具有与 ASP.NET MVC 特性相同的名称和行为，但它是在一个不同的程序集中定义的。归根结底，它是一个完全不同的类，但具有相同名称和大部分相同的行为。

10.2.3　使用 Web API

构建在一个 ASP.NET MVC 应用程序内部的 Web API 模块是由该 ASP.NET MVC 应用程序托管的。要使用 Web API 模块，你只需要使用 ASP.NET MVC 站点即可。在这一点上，你所看见的都是你用已有 API 能够使用的 HTTP 端点，其中包括一个基于 JavaScript 的 API。

1. 从 JavaScript 调用 Web API

从 Web 客户端内部，Web API 前端就是一个简单的 HTTP 端点集合。可以使用 jQuery 工具来放置 HTTP 调用，或者使用 KnockoutJS 工具来呈现出对象集合。以下是你需要用来设置 Web API 远程数据绑定的代码(这是一个.cshtml 视图文件的摘要)：

```
<h1>Web API DEMO</h1>
<script type="text/javascript">
    function News(title) {
        this.title = title;
    }
    function NewsViewModel(listOfNews) {
        this.allNews = listOfNews;
    }
</script>
<script type="text/javascript">
    $(document).ready(function () {
        getNews(function (listOfNews) {
            ko.applyBindings(new NewsViewModel(listOfNews));
        });
    });

    function getNews(callback) {
        $.ajax({
            url: "/api/news/all",
            type: "GET",
            statusCode: {
                200: function (listOfNews) { callback(listOfNews); },
                404: function () { alert("No news found!"); }
            }
        });
    }
</script>

<div id="news-container">
    <h2>Late breaking tennis news</h2>
    <table>
```

```
        <thead><tr>
            <th>Title</th>
        </tr></thead>
        <tbody data-bind="foreach: allNews">
            <tr>
                <td data-bind="text: Title"></td>
            </tr>
        </tbody>
    </table>
</div>
```

图 10-3 显示了最终的结果。

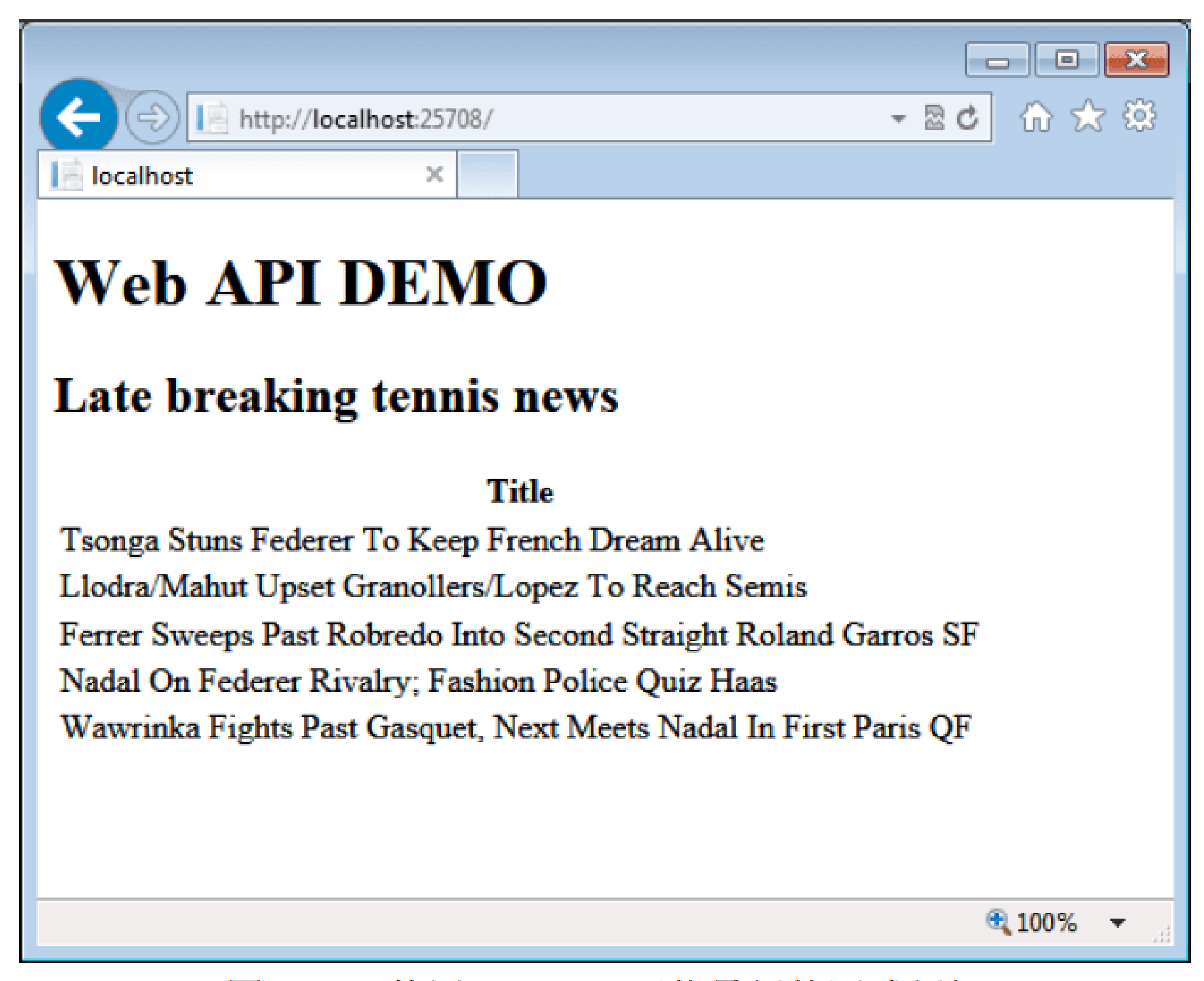


图 10-3　使用 Web API 下载最新的网球新闻

$.ajax jQuery 工具让你能够指定 HTTP 动词和标头。这样一来，你就能够轻易地作好调用 Web API 后端上的任意类型操作的调用准备。

2. 从服务器端代码调用

由于 Web API 模块是一个普通的 HTTP 前端，因此从一些.NET 服务器端代码内部调用它就与从任何其他类型的远程端点调用没什么区别。从 ASP.NET MVC 应用程序内部，可以使用控制器方法来生成 HTML，如下所示：

```
public ActionResult News()
{
```

```
        var client = new WebClient();
        var content = client.DownloadString("/api/news");
        var serializer = new JavaScriptSerializer();
        var listOfNews = serializer.Deserialize<IList<News>>(content);
        return View(listOfNews);
    }
```

该代码会将数据作为 JSON 下载，然后使用 JavaScriptSerializer 类将其转换到一个.NET 对象中——具体来说，是一个 News 数据对象列表。该列表随后会作为视图模型传递给视图引擎以通过 Razor 生成用户界面。

```
@model IList<Simplest.Models.Dto.News>
<h1>Web API DEMO</h1>

<div id="news-container">
    <h2>Late breaking tennis news</h2>
    <table>
        <thead><tr>
            <th>Title</th>
        </tr></thead>
        <tbody>
            @foreach (var n in Model)
            {
                <tr>
                    <td>@n.Title</td>
                </tr>
            }
        </tbody>
    </table>
</div>
```

上述的代码使用了最简单(但却非常有效)的下载和转换 JSON 数据的方法。在最新版本的.NET 框架中，可以使用其他内置的 JSON 转换器或 HTTP 客户端类，它们提供了对异步调用的更好支持。但是最终，无论使用何种特定工具或技术，其目的都是下载 JSON 数据并将其转换成.NET 对象。

3. 进行异步调用

我们都知道编写的代码必须总是高度可扩展的。现在，我可以惬意地承认，没有任何代码存在于高度可扩展应用程序的上下文中。然而，当面临为可扩展性重构时，在读取和写入方面有两件基本的事情可以完成：使用缓存和异步逻辑。长时间以来，在.NET 语言中编写异步代码都很麻烦。但是，从 C# 5.0 和.NET Framework 4.5 开始，像 async/await 这样的新关键

字通过添加少许语法使得这项任务变得简单了。

这样一来，虽然我仍然不认为出于我们都需要可扩展性的原因，所有的远程调用都必须是异步的，但我也没发现特殊的原因不去编写带有 async/await 关键字的异步远程调用，这些调用使得我们可以避免无阻塞调用(特别是来自用户界面的)并且仍能设法保持代码的高可读性。下面是如何通过使用 async/await 语言工具重写 Web API 方法的代码：

```
public async Task<IList<News>> GetAll()
{
    var url = ...;
    var client = new HttpClient();
    var rss = await client.GetStringAsync(url);
    var news = ParseRssInternal(rss);
    return news;
}
```

作为使用 WebClient 进行远程调用的替代选项，可以使用最新的 HttpClient，它是老式可靠的 WebClient 客户端的修改及更丰富的版本。HttpClient 是在 System.Net.Http 名称空间中定义的。

ASP.NET MVC 方法负责调用使用以下代码进行异步编码的 Web API 模块：

```
public async Task<ActionResult> News()
{
    var feed = new FeedController();
    var model = await feed.GetAll();
    return View(model);
}
```

从 JavaScript 调用一个异步编码的 Web API 模块不需要对客户端代码作任何修改。

10.2.4　设计面向 RPC 的接口

到目前为止，基于 RESTful 能够解决公共 API 定义问题的假设，我们已经尝试了使用 Web API。总的来说，对于 ASP.NET MVC 的开发人员，正确地看待 Web API 是最困难的事情。Web API 看起来与你已经习惯的普通控制器类似，并且轻率地将其归结为又一种奇怪类型的控制器的误解还是很多的。

Web API 就是你业务领域的 SDK；如果用在 ASP.NET MVC 应用程序中，它就是你应用程序要基于的 SDK。但是同时，此 SDK 是通过 HTTP 来进行公共公开的。

1. HTTP 上的 CRUD 只是选项之一

API 上默认启用的约定倾向于让开发人员从 HTTP 上 CRUD 的上下文角度来看待 API。可以定义一个模型对象并构建一个 API 控制器来在其上运行。API 方法是根据对指定资源执

行基本操作的基础 HTTP 动词来建模的。这就为设计提供了 RESTful 化(RESTfulness)的风格。

正如之前所看见的，此方式也会造成命名冲突，因为你会发现很难对一个返回遵循非常规结构的数据的方法命名。这种 RESTful 方式并非唯一可行的方式：面向 RPC 的方式也可行。

很难说 REST 是否优于 RPC。但我认为应用程序的最大责任就是面向 RPC。例如，大多数 ASP.NET MVC 站点都是建在方法集上的，这些方法只能通过 GET 或 POST 调用。实际上，ASP.NET MVC 刚推出的几年里，许多个人项目就已经开始让 ASP.NET MVC 表现出明显的 RESTful 化风格了。

不过，最后的解决方式很简单：无论 Visual Studio 中 Web API 项目的默认约定是什么，你都可以获得 API 的完全控制，并且如果喜欢的话，可以赋予其 RPC 风格。

2. 操作特性

将 Web API 的设计转变成 RPC 的关键是使用操作特性。与你在 ASP.NET MVC 控制器中所做的非常类似，可以使用像 HttpGet、HttpPost、HttpPut 或 HttpDelete 这样的特性来标记 Web API 方法，这样一来，不管名称是什么，都可以考虑使用这些特性，只要请求动词可以匹配某特性即可。

```
[HttpGet]
public IEnumerable<News> All()
{
   ...
}
```

All 方法现在可以用于 GET 请求，即便其名称并非以 Get 开头。Web API 库提供了预定义的特性，比如 HttpGet、HttpPost、HttpPut 或 HttpDelete，就像 AcceptVerbs 一样可以用来将一个或多个 HTTP 动词绑定到方法。

3. 自定义路由

仅仅使用操作特性不足以让你获得 API 的全面控制。你可能还需要修改在 Visual Studio 向导为你创建的初始标准 Web API 项目中定义的默认路由。默认路由并不具有用于操作名称的占位符。你可能会想要将之前在“路由到操作方法”一节中讨论过的 DefaultApi 路由替换成下面这样：

```
RouteTable.Routes.MapHttpRoute(
              name: "RpcRoute",
              routeTemplate: "api/{controller}/{action}/{id}",
              defaults: new { id = RouteParameter.Optional });
```

主要的区别是，现在路由为方法名称保留了一个占位符——{action}占位符。这使得处理

请求到方法的过程几乎与 ASP.NET MVC 普通控制器相同，除了该 URL 中额外的 api 这几个字。

```
http://server/api/news/all
```

上面的 URL 将在 RESTful 模型中产生一个异常，但将 All 方法上的 HttpGet 与定义的自定义 RpcRoute 路由结合使用，它就能正常运行了。

4. 特性路由

Web API 的大部分设计都归功于 ASP.NET MVC，即使它与 ASP.NET MVC 运行时完全没有联系。在随 Visual Studio 2013 和 ASP.NET MVC 5 同时到来的 Web API 的最新版本中，你会发现一个称为特性路由的功能，它在 ASP.NET MVC 没有直接对应项；相反，它派生于过时的 WCF Web HTTP 绑定。

经典的路由是基于约定的。当一个请求传入时，会针对 global.asax 中注册的路由模板来匹配 URL。如果匹配成功，则为请求提供服务的合适控制器与操作方法就能从模板中确定。如果匹配不成功，则请求将被拒绝并且其结果通常是一个 HTTP 404 消息。路由匹配只会基于首次匹配生效。首次匹配的最直接后果就是，路由注册的顺序很重要。大多数特定路由应该出现在列表前面；笼统及最通用的路由应该位于列表最后。这样做的重大意义何在呢？

在具有强烈 REST 风格的中型应用程序中已存在的路由数量可能会非常多，轻易就能达到数以百计。要确定超过 200 个路由的正确顺序肯定会变得非常困难，而且你会陷入围绕在自动化测试期间检测到的或者由用户和测试人员通报的回归周围的无穷循环之中。特性路由提供了一种顺畅方式来处理这种情况中的路由。

顾名思义，特性路由就是让路由(作为特性)附加到特定操作方法，如以下代码所示：

```
[HttpGet("orders/{orderId}/show")]
public Order GetOrderById(int orderId)
{
   ...
}
```

上述代码片段表明，只要请求 URL 模板匹配指定模式，GetOrderById 方法就可通过 HTTP GET 进行调用。路由参数——orderId 令牌——必须匹配在方法签名中定义的一个参数。还有另外一些详细信息需要讨论，但特性路由的要点都已在这里作过介绍了。不可否认，特性路由与现在已过时的 WCF Web HTTP 编程模型尤其是 WebGet 特性非常类似。

```
[WebGet(UriTemplate="orders/{id}/show")]
Order GetOrderById(int id);
```

5. 开启特性路由

默认情况下特性路由没有开启，但它可以与常规路由同时运行。下面是开启它的标准方法：

```
public static class WebApiConfig
{
    public static void Setup(HttpConfiguration config)
    {
        config.MapHttpAttributeRoutes();
    }
}
```

如果打算将这两种类型的路由一起使用，建议你优先使用特性路由。这意味着你要在开始将全局路由添加到系统前调用 MapHttpAttributeRoutes。

可以通过特性为每个方法定义一个想要的路由，并且还可以为用于调用操作的 HTTP 方法定义筛选器。在 Web API 中，像 HttpGet、HttpPost、HttpDelete、HttpPut 这样以及所有其他的特性类都已经用一个重载扩展了，以便接收路由 URL 模板。如果愿意，还可以使用 AcceptVerbs 特性，其第一个参数表示方法名称，而第二个参数设置路由，如下所示：

```
[AcceptVerbs("GET", "orders/{id}/show")]
```

你还可以将 AcceptVerbs 用于不受支持的 HTTP 方法或者 WebDAV 方法。

特性路由还可以使用与经典路由稍许不同的语法来支持参数约束。该 API 具有一个预定义的约束列表，比如 int、bool、alpha、min、max、length、minlength 以及 range。这里显示了如何使用它们：

```
[HttpGet("orders/{id:int}/show")
```

约束的目的显而易见。要了解更多详细信息，可以查阅 http://www.asp.net/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2 处的一些文档。

可以随你所愿将许多约束串联使用，也可以创建自定义约束。

```
[HttpGet("orders/{id:int:range(1, 100)}/show"]
```

要创建一个自定义约束，需要创建一个实现 IHttpRouteConstraint 接口的类并使用以下代码来注册它：

```
var resolver = new DefaultInlineConstraintResolver();
resolver.ConstraintMap.Add("custom", typeof(YourCustomConstraint));
config.MapHttpAttributeRoutes(new HttpRouteBuilder(resolver));
```

IHttpRouteConstraint 接口只有一个名为 Match 的方法。

10.2.5　安全性考量

总体上，Web 应用程序的安全控制要比 Web API 模块的安全控制简单，这只有一个原因：要考虑的情况较少。由于 ASP.NET MVC 应用程序是针对最终用户的，因此很大程度上，安全性指的是对用户进行身份验证并确保每个已验证用户都有权限执行指定操作。

表单验证(例如，在其中输入凭据的登录窗口)是最常用的捕获和验证凭据的方式。在 ASP.NET MVC 中，操作方法上的 Authorize 特性会指示运行时只有已验证用户才能调用该方法。Authorize 特性上的参数或该方法实现中的普通代码随后将执行更为复杂的检查，以判定分配给指定登录用户的角色。ASP.NET MVC 应用程序安全性方面的内容在第 6 章“应用程序安全性”中已作过详细介绍。

谈及 Web API 时，还有一些额外的场景需要注意。Web API 模块就是一个 API，可以为第三方开发人员的使用而编写它。那么你要如何控制该客户端是已知并通过授权的呢？

1. 宿主要负责处理安全性

Web API 模块面临的最简单但并非最常见的场景是，API 和宿主都由同一个团队管理。在这种情形下，宿主要负责对用户进行身份验证并且避免出现任何可能的调用 API 的用户界面。

因此，Web API 会假设所有身份验证都发生在宿主内；例如，在 IIS 内部通过 HTTP 模块对内置模块或自定义模块进行身份验证。为了应对这种情况，需要在 API 控制器类或单独方法上使用 Authorize 特性。这与在普通 ASP.NET MVC 应用程序中面对的情况几乎相同。

```
[Authorize]
public IList<News> GetAll()
{
   ...
}
```

注意这里使用的 AuthorizeAttribute 类与 ASP.NET MVC 中使用的 AuthorizeAttribute 不同。虽然编程接口相同，但这个类是在另外的位置定义的，以避免对 ASP.NET 运行时的依赖。

你还可以将 Authorize 特性设置成全局的，以便它能应用于所有 API 控制器的所有方法。如果打算这样做，请使用以下代码：

```
public static SetupAuthorization(HttpConfiguration config)
{
    config.Filters.Add(new AuthorizeAttribute());
}
```

请牢记，在这种情况下所有的方法都是要进行安全控制的对象；不过，AllowAnonymous 特性会让你为匿名调用保留一个可用方法。

```
public class NewsController : ApiController
```

```
{
    [AllowAnonymous]
    public IList<News> GetAll() { ... }
    public HttpResponseMessage PostNews(News news) { ... }
}
```

在方法主体中进行身份验证之后，可以使用 ApiController 类上的 User 属性来检查详细信息以及用户的角色，并且如果未通过检查就拒绝该调用。

2. 使用基本身份验证

在宿主为身份验证和授权规则提供的方案之外，就是在 Web API 模块中集成安全层。最简单的方式是使用构建在 Web 服务器中的基本身份验证。基本身份验证是基于用户凭据封装在每一个请求中这一思想的。

基本身份验证有其优缺点。从优点方面来说，它支持主流浏览器；它是一项互联网标准；它易于配置。其缺点是，凭据是随每个请求发送的，并且更糟的是，它们都是以明文发送的。

基本身份验证预期凭据会被发送到服务器上验证。随后当凭据是有效时，请求会被接受。如果请求中没有凭据，就会显示一个交互式对话框。实际上，基本身份验证还需要一个自定义 HTTP 模块来根据存储在某自定义数据库中的账户检查凭据。

注意：

将基本身份验证与一个执行自定义凭据验证的层结合起来使用是简单且十分有效的。要克服凭据作为明文发送这一限制，你应该总是在 HTTPS 上实现基本身份验证解决方案。

3. 使用访问令牌

此处的思想是，Web API 会接收一个访问令牌(通常是一个 GUID 或一个字母数字字符串)、验证它、并在该令牌未过期且对应用程序有效时为请求提供服务。有各种方式和不同的安全解决方案来颁发令牌。

令牌的最简单方案是，当客户联系你要使用你的 API 时在线下颁发令牌。你要创建令牌并将其关联到特定客户。此后，该客户就要对滥用或错用 API 负责。

Web API 后端需要有一个检查令牌的层。可以将这个层作为普通代码添加到任意方法，或更好的是，将其配置成一个消息处理程序。在 Web API 中，一个消息处理程序就是一个组件，该组件会检查 HTTP 请求并设置该请求上的主体，以便 ApiController 上的 User 属性能被正确填充。

注意：

类似于 ASP.NET 的 HTTP 处理程序，消息处理程序仅会应用于通向 Web API 的通信流。

可以在 http://www.asp.net/web-api/overview/working-with-http/http-message-handlers 阅读到更多内容。

4. 使用 OAuth

一个更加复杂的方案是使用 OAuth 身份验证来限制对 Web API 模块的访问。第 6 章讨论了如何将 OAuth 与 Facebook 或 Twitter 结合起来使用以对站点的用户进行身份验证。这里的关键在于，将你的 Web API 模块转变成 OAuth 服务器，类似于 Twitter 或 Facebook 所做的。

归根结底，OAuth 就是之前讨论过的令牌方案的一种形式。不同之处是，你需要使用一个不同的组件来在检查了用户的凭据之后在线颁发令牌。授权服务器介于受保护资源(Web API)和潜在客户之间。图 10-4 显示了典型 OAuth 会话的布局。

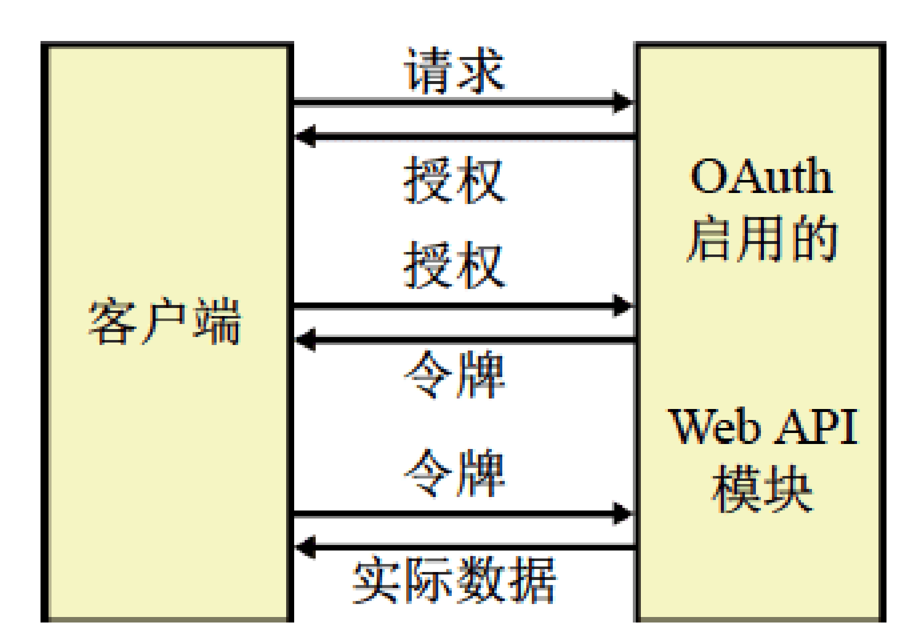


图 10-4　客户端与 Web API 服务器之间典型的 OAuth 握手

你要如何在实践中对其编码呢？

这很大程度上取决于你，但它很可能需要使用 Web 后端以用作授权服务器。它将根据由已验证用户提供的凭据(比如 Facebook 提供的)来负责颁发令牌。接着，你要使用 Web API 层中的消息处理程序来处理该令牌并设置主体以真正对调用授权。在该消息处理程序中，你还可以引入任意的机制来以你认为有意义的方式使该令牌无效——这些方式包括过期、误用、或因为调用方的调用超出了分配配额。

5. 使用跨源资源共享

跨源资源共享(CORS)是一个 W3C 标准，它定义了想要对不同域进行 Ajax 请求的网页行为。CORS 放宽了所有浏览器都实现的同源策略，该策略会将调用限制为只能在发出调用页面所在的单个域中进行。

注意：

浏览器实现了同源策略并出于安全原因会阻止跨域 Ajax 调用。因此，CORS 是否就是一种值得信赖的方式呢？或者，它不会削弱安全性吗？无论目标站点是否允许接受跨域调用，浏览器的同源策略都会限制客户端页面进行任何跨域调用。CORS 规范只是定义了一个协议，通过该协议，站点就能允许接收来自远程站点的调用。CORS 不会削弱安全性，这是因为跨

域调用会查找双方是否都允许此类调用。

你需要通过一个新的名称为 EnableCors 的特性来启用 CORS。可以像方法一样在控制器上设置该特性。要让该特性生效，你还需要调用 global.asax 中 HttpConfiguration 对象上的 EnableCors 方法。

```
using System.Web.Http.Cors;
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.EnableCors();
    }
}
```

你要将新创建的 EnableCorsAttribute 类的实例传递到你得到的方法以便为所有方法和所有控制器全局开启 CORS。如果使用在控制器级别开启的 CORS 并想要在一个特定方法上关闭它，只需要使用 DisableCors 特性即可。

10.3　协商响应格式

在你使用示例 NewsController 类进行一些实践之后，你大概会注意到，默认情况下所有操作方法都会返回其序列化为 JSON 字符串的数据。这正是大多数时候开发人员想要的，但它并不适合于所有可能的情形。实际上，有时候你还会想要通过 XML、普通文本或者 iCalendar 源(这当然可行)来公开相同的数据。这是否意味着你需要为每个可行的输出格式使用多个操作方法呢？这正是 Web API 中内容协商所适用之处。

10.3.1　ASP.NET MVC 方式

在经典的 ASP.NET MVC 中，可以通过在每个操作方法中手动强制进行有序分离来处理这一问题，该有序分离介于产生要返回的原始数据的代码和以适合调用方的方式格式化原始数据的代码之间。

1. 理解请求格式

首先，在调用方能够指定预期的输出格式方面，ASP.NET MVC 并不具有默认和明确定义的方法。一个广泛接受的解决方案包括，使用一个查询字符串(或特设路由)中的参数，该参数会将所选的响应格式带入控制器代码内部。这里是我最喜欢的结构：

```
public ActionResult All(Boolean xml = false)
```

```
{
    // Get raw data
    var listOfNews = NewsHelper.GetAllNews();

    // Serialize raw data
    if (xml)
        return new NewsXmlFormatter().Serialize(listOfNews);
    return Json(listOfNews, JsonRequestBehavior.AllowGet);
}
```

这意味着，通过将 xml=true 添加到 URL 的查询字符串，你就能让数据作为 XML 返回。否则，你将得到作为 JSON 返回的数据。对于以 XML 架构格式化数据来说，根据所处环境不同，我会借助于一个通用的 XML 格式化器或者一个特定对象的特定格式化器。

如果不喜欢使用参数，则可以选择一个特设的路由，比如以下这个：

```
routes.MapRoute(
    name: "Xml",
    url: "xml/{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id =
        UrlParameter.Optional, xml = true }
);
```

更普遍的情况是，找出一种方式得到预期的响应格式，这项任务取决于开发人员。使用路由或参数都只是可行的方法而已。可以要求调用方添加一个特定的 HTTP 标头，或者可以检查一些默认的 HTTP 标头，比如 Accept 或 Content-Type。

2. 强制将数据与格式分离

正如可以从之前的代码中看出的，ASP.NET MVC 控制器方法的主体通常分为两部分：操作结果的获取，以及操作结果的处理。是否将这两个阶段分离取决于你。只要你仅在 JSON 和 XML 之间做选择，该问题就不难处理。不过，如果有多个格式要处理又怎么办呢？而且，如果服务已经位于生产环境中，但你必须为所有方法添加一个新格式，这时又该怎么办呢？

此时就需要一个更好的策略以用于响应格式的协商，并且 Web API 中已经提供了该策略。

10.3.2　内容协商是如何在 Web API 中运行的

在 Web API 中，内容协商一词指的是检查传入请求结构(通常是 HTTP 标头)的处理过程，以便确定客户端想要接收的一个(或多个)理想响应格式。

1. 涉及的 HTTP 标头

在 Web API 中，有两个 HTTP 标头在内容协商中扮演了关键角色。它们就是 Accept 与

Content-Type。这两者中，起主要作用的是 Accept，这是因为 Content-Type 只有在 Accept 标头缺失或具有无效内容的情况下才会起作用。以下代码显示了如何在调用 Web API 端点的 jQuery 调用中设置 Accept 标头以接收一些返回的 XML。

```
$.ajax({
     url: "/api/news/all",
     type: "GET",
     headers: { Accept: "text/xml; charset=utf-8" }
});
```

在 C#代码中，你要像下面这样设置 Accept 标头：

```
var client = new WebClient();
client.Headers.Add("Accept", "text/xml; charset=utf-8");
```

ApiController 类内部具有一段代码以处理由每个方法返回的数据并将该原始数据转换成合适的 JSON 或 XML。原始结果的生成与格式化之间的有序分离在 Web API 中是原生的，并且作为开发人员，可以选择 XML 或 JSON 而无须对嵌入式格式化器组件的细节进行公开。通过使用 Web API，你就可以相应地对 JSON 或 XML 使用自动化和自由序列化。

如果缺少 Accept 标头，或者具有一些无效内容，Web API 就会检查 Content-Type 标头。如果该标头具有有效内容，则其内容就会用来格式化结果。如果通过 Accept 或 Content-Type 不能确定有效的序列化格式，Web API 就会选取注册到运行时的第一个格式化器。默认情况下，它是 JSON 格式化器。

2. 修改默认格式化器

谈及将返回数据格式化成 JSON 的服务时，一个常见问题就是在 JavaScript 客户端和.NET 客户端中使用时，对象属性名称的大小写问题。在.NET 和 JavaScript(以及相应程度的 Java)中，默认使用的是骆驼命名法(camelCasing)。然而，在单纯的.NET 中，默认使用的是帕斯卡命名法(PascalCasing)。由于你是在 C#中编写服务，因此你很可能会使用帕斯卡命名法来定义数据传输对象(DTO)。而当此数据被 JavaScript 端接收时就会产生冲突。

作为替换默认格式化器的示例，让我们从提供特殊类型格式化器的各种社区项目中选取一个。我使用了 JsonCamelCaseFormatter，可以在 https://gist.github.com/rdingwall/2012642 处找到它。这个类修改了 JSON 数据序列化的方式，以确保将骆驼命名法应用到成员名称。

自定义格式化器就是一个继承自 MediaTypeFormatter 的类的实例，并且重写了 4 个关键方法：CanReadType、CanWriteType、ReadFromStreamAsync 以及 WriteToStreamAsync。这些方法的名称能够一目了然地表明其用途。这里是你怎样才能注册新的 JSON 格式化器的代码：

```
public static class WebApiConfig
{
```

```
        public static void Register(HttpConfiguration config)
        {
           var index = config.Formatters.IndexOf(config.Formatters. JsonFormatter);
           config.Formatters[index] = new JsonCamelCaseFormatter();
        }
    }
```

尤其是，该示例显示了如何用 JsonCamelCaseFormatter 的实例替换默认的 JSON 格式化器——JsonMediaTypeFormatter 的一个实例。如果只想将另一个格式化器添加到该列表，可以使用 HttpConfiguration 对象的 Formatters 集合上的 Insert 或 Add 方法。

3. 为特殊类型定义格式化器

另一个我经常遇到的常见情形是，面临返回一些类型的不同格式的需求，尤其是不同的 XML 架构。前面提到的 CanReadType 和 CanWriteType 方法都能为此提供一些帮助。以下是 News 类型的一个自定义 XML 格式化器：

```
public class NewsXmlFormatter : MediaTypeFormatter
{
    public NewsXmlFormatter()
    {
        SupportedMediaTypes.Add(new MediaTypeHeaderValue
                                   ("application/xml"));
        SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/xml"));
    }

    public override bool CanReadType(Type type)
    {
        return false;
    }

    public override bool CanWriteType(Type type)
    {
        var result = (type == typeof(News) || type == typeof(IList<News>));
        return result;
    }

    public override Task WriteToStreamAsync(Type type,
                                            Object value,
                                            Stream writeStream,
                                            HttpContent content,
                                            TransportContext transportContext)
    {
        return Task.Factory.StartNew(
```

```
            () =>
              {
                  var listOfNews = (IList<News>) value;
                  using (var writer = new StreamWriter(writeStream))
                  {
                      foreach (var n in listOfNews)
                      {
                          writer.WriteLine("<news>");
                          writer.WriteLine("<title>{0}</title>", n.Title);
                          writer.WriteLine("</news>");
                      }
                  }
              });
    }
}
```

格式化器仅会处理序列化，并忽略反序列化(参见图 10-5)。

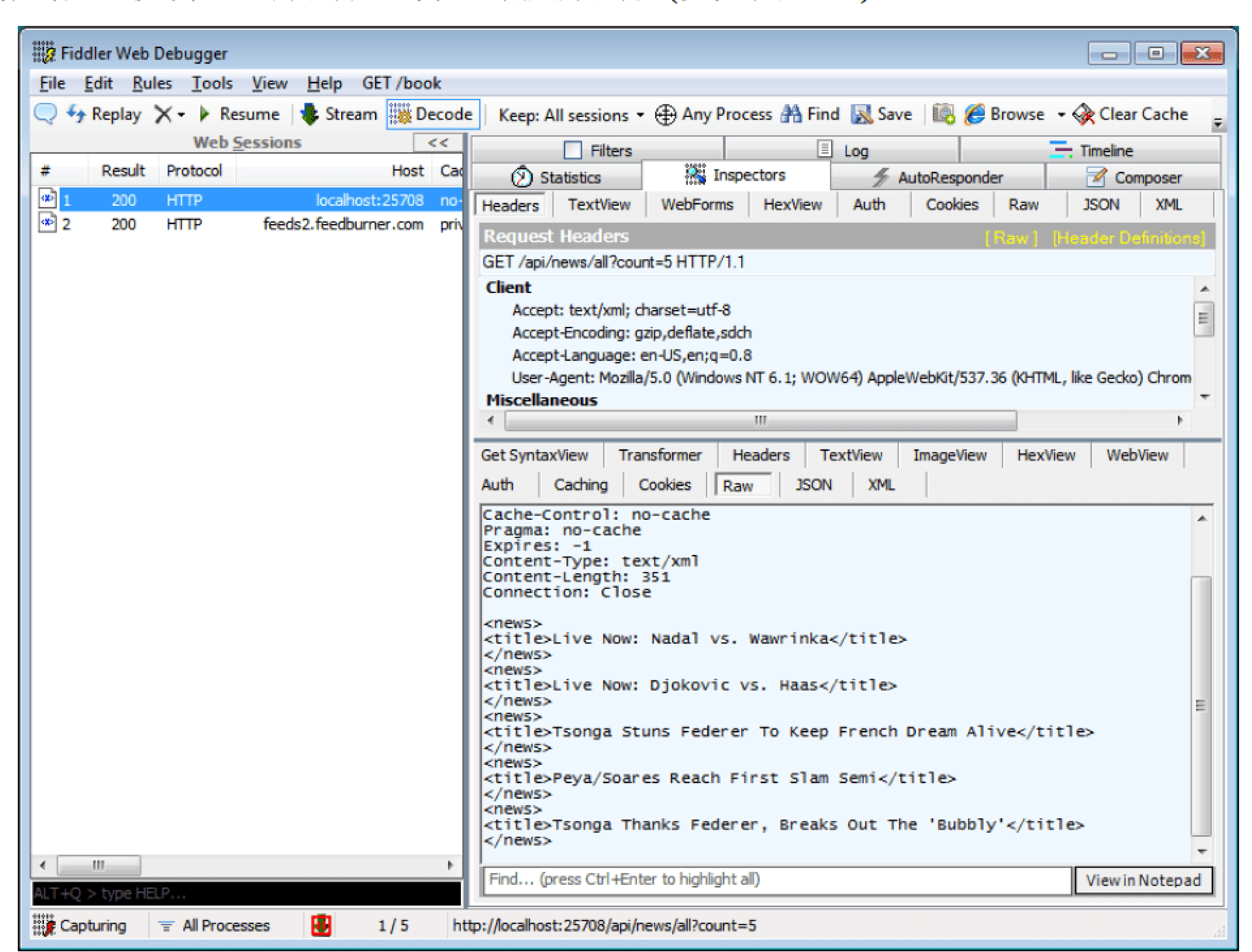


图 10-5　通过自定义 XML 格式化器为 XML 数据请求提供服务

现在，问题就变成了你如何能确保该格式化器的调用仅用于特定类型，并且默认的格式化器用于其他情况。这主要取决于 Web API 运行时注册格式化器的顺序。

```
var xmlIndex = config.Formatters.IndexOf(config.Formatters.XmlFormatter);
config.Formatters.Insert(xmlIndex, new NewsXmlFormatter());
```

上面的代码会将特定类型的 XML 格式化器插入到通用格式化器之前。这样一来，它就能被首先调用了。

10.4　本章小结

本章主要介绍了短期内会在 Web 解决方案中扮演重要角色的框架。正如第 12 章“让网站对移动端友好”中会明确指出的，现今，Web 解决方案有时是由混合的静态 HTML 标记组成的，该混合标记带有由运行中的来自远程后端的数据生成的动态节。该远程后端的特征一直都在不断变化。它最初是基于 SOAP 的 Web 服务，然后是 WCF 服务，再之后就是 ASP.NET MVC 应用程序的一部分。

Web API 总是与 ASP.NET MVC 控制器一样精简且易于编码。不过，不仅如此，Web API 还不依赖于 ASP.NET 框架并且甚至能够在 IIS 之外托管；例如托管在 Windows 服务中。

Web API 是一个成熟的框架；它应该有一本恰如其分地叙述其所涉及内容的书。本章从 ASP.NET MVC 开发人员的角度简要介绍了其经典轮廓。Web API 是一个设计选项，但如果只是要设置一个网站的话，我不认为你会需要使用它。如果你考虑的正是这种情形，那么 Web API 很可能对你毫无用处。但如果预见了用于一些核心 SDK 的多个前端，那么 Web API 就能起到很大帮助，因为这正是其被特别创建的目的。

第 III 部分

移动客户端

第 11 章

有效的 JavaScript

一个人的出身并不重要，重要的是他成长为什么样的人。

——*J. K. Rowling*

JavaScript 是在 20 世纪 90 年代为 Web 而开发的，如今 JavaScript 已经广泛用于 Web 应用程序之外的领域。例如，你会发现 JavaScript 被用于为 Adobe Photoshop、谷歌 Chrome 和 Mozilla Firefox 编写应用程序扩展。它也被用于创建本地移动应用程序；比如 PhoneGap 或 Appcelerator Titanium。JavaScript 还被用于某些类型的服务器端应用程序，最引人注目的就是 Node.js。

JavaScript 的命运很奇特。它作为首批 Web 粉丝的语言而诞生，从 20 世纪 90 年代初期开始经历了一系列的高峰和低谷。在某种程度上，它的使用受到了动态 HTML 出现的推动，但随着与 Java Server Pages 和 ASP.NET 的整合又在短短几年内沉寂了，因为 Web 开发的重点又重新回到了服务器端。随着 Ajax 的使用，JavaScript 再次恢复了生机，并且 Node.js 和移动框架的出现强化了它的生命力。正如本章开头那句格言提醒我们的那样，JavaScript 的出生并不重要；它的成长和发展才更具影响力。

JavaScript 是一种很容易上手的编程语言；但同时，对它的掌握不容小觑。

如今所有的 Web 应用程序都被要求添加越来越多的客户端功能。这不仅是指客户端的验证和工具，而是指整个 Ajax 应用程序。一个完整的 Ajax 应用程序是将页面交互减少到最低限度的 Web 应用程序。一个完整的 Ajax 应用程序集中于极少的几个核心页面，这些页面的用户界面随着用户的操作以交互方式进行变更。显然，在 HTML 文档对象模型(DOM)上做的大量工作都需要使用 JavaScript。

注意：

作为 ASP.NET MVC 开发人员，你要准备好编写越来越多的 JavaScript 代码。更重要的是，你要准备好导入别人编写的越来越多的 JavaScript 代码。这两者并不是互斥的。实际上，现在几乎没有哪个网站的代码编写不从 jQuery 和其他一些库，如 AngularJS、KnockoutJS 或

Modernizr，导入功能了。

11.1　重温 JavaScript 语言

JavaScript 是一种不寻常的语言，因为它是为非开发人员所创建；其进入门槛很低，又足够灵活，专家可以用它来做他们想做的几乎一切事情。依我之见，如今所面临的挑战是，普通的 JavaScript 开发人员如何用一些优秀的设计来创建有效的内容，同时将可读性和可维护性保持在适当的水平上。为此，本节旨在重新整理 JavaScript 的一些基本情况。

11.1.1　语言基础知识

JavaScript 是解释型代码，意即 JavaScript 程序需要一个环境以便在其中运行。它们的天然运行环境是 Web 浏览器。该语言的语法由一种健壮的标准 ECMA 262 所驱动，它也被称为 ISO/IEC 16262:201。经过这么多年的发展，产生了一些基于该标准的不同实现的 JavaScript 语支。有时候，这些语支有着完全不同的名称，如 JScript、 JScript.NET、 ActionScript 和 EcmaScript。但归根结底，都是由不同引擎处理 JavaScript 源代码的结果。经常与 Web 浏览器相关联的流行引擎有 V8(谷歌 Chrome)、Chakra(微软 Internet Explorer)、Gecko(Firefox)和 Opera。

让我们简要地浏览一下该语言的基础知识，并重新整理一下被大多数 ASP.NET 开发人员拿起来在用但从未深入研究的一些概念。

1. 类型系统

JavaScript 类型系统包括基本类型和几个内置对象。不过当你编写 JavaScript 代码时，可以使用的类型范围实际上更大。除了使用内置的对象，还可以依赖由宿主提供的对象。典型的例子就是最常见的 JavaScript 宿主——Web 浏览器——发布到 JavaScript 引擎中的窗口对象。

基本类型是 number、string、Boolean、null、undefined、Object 和 function。内置的对象是 Array、Math、Date、Error、RegExp，以及几个用于基本类型的封装对象：string、Boolean 和 number。

number 类型表示带有 0 或更多小数位数的浮点数。没有用于整数、长整数或字节的单独类型。数字的范围介于-10^{308} 和 10^{308} 之间。可以写入十进制、八进制或十六进制格式的数字。特别的数字 NaN 表示没有意义的数学运算结果(如被零除)。

string 类型表示具有 0 个或更多个字符的行；它并不表示一个数组。单个字符由一个字符的字符串表示。字符串中的特殊字符以反斜杠(\)开头，比如\n 表示一个回车符。字符串的内容要放进一对单引号或双引号内。基本值被封装在添加了几个方法的字符串对象中，这些方法包括 split 和 substring。

2. Null 对比 undefined

不包含有意义值的 JavaScript 变量可以被分配为 null 值和 undefined 值。两者的区别是什么呢？就空值而言，JavaScript 采用了一个像 C#和 Java 这样的高级语言都会错过的细微差别。

具体来说，undefined 是运行时环境分配给所有正在声明的变量的默认值。最引人注意的是，未赋值的变量包含 undefined，但不包含 null 或某些默认值，这和在 C#中一样。另一方面，null 仍是一个表示 null、空或不存在引用的值，但它是由开发人员显式分配的。简单地说，一个设置为 undefined 的变量不会被任何代码触及到；而包含 null 的变量会通过你代码中的一些路径被分配到某个值。

如果在未赋值的变量上运行 typeof，就会得到 undefined——它本身就是一种独特的类型。如果在被赋予 null 值的变量上运行 typeof，你会获得 object。注意下面的代码：

```
var x;     // hence, undefined
var y = null;
```

如果比较一下 x 和 y 会发生什么？如果使用==运算符来比较，会得到 true，即 undefined 的最终计算结果为 null。如果使用===运算符比较，会得到 false：两个变量持有相同的值，却是不同的类型。

3. 局部变量和全局变量

在 JavaScript 中，变量是一个并不局限于总是存储固定类型值的存储位置。被分配了一个值后，变量就会承接被存储的数据类型。因此，JavaScript 变量可能在其生命周期中几次更改它的类型。

```
var data = "dino";  // now data is of type string
data = 123;         // now data is of type number
```

这种行为有别于 C#变量的典型行为，除非 C#变量被定义为.NET 4 中的动态类型。在首度使用时变量就被赋予了生命；在这之前，它们只会具有一个 undefined 值。

在定义变量时，你应一直使用 var 关键字作为对解析器和你自己的提示。var 关键字并非绝对必要，但强烈建议使用。如果通过使用 var 对变量进行了声明，那么在函数内定义的变量就会仅限于这个函数。如果没有声明，变量会被视为全局的，但它们会保持 undefined，直到函数执行一次。在全局范围中声明的变量都是全局变量，无论是否使用 var。

```
<script type="text/javascript">
  var rootServer = "http://www.expoware.org/";   // global
  section = "mobile";                            // global
</script>
```

```
<script>
  function doSomething() {
     var temp = 1;                         // local
     mode = 0;                            // global, but undefined until called
  }
</script>
```

JavaScript 运行时环境会将全局变量存储为通过 this 关键字引用的隐藏对象的属性。浏览器往往通过窗口对象反射该对象。

在所有的编程语言中，能够使用全局变量都会使编码变得更容易。然而，全局变量也有负面影响。其主要缺点是，在你的代码、第三方库、广告合作伙伴和分析库的不同部分中所定义的变量之间存在着名称冲突的风险。名称冲突加上 JavaScript 变量的动态类型可能会导致你无意中修改了应用程序的状态，进而在运行时产生令人不快的异常。

需要注意很容易就会无心地创建出全局变量：缺失 var 的话，你最终获得的就是全局变量；在赋值中键入错误的变量名称，也会得到一个新的全局变量。这后一种特点是很可能出现的，因为在 JavaScript 中，可以在无须首先声明变量的情况下使用它。当你需要使用全局变量时，一个好的技巧是将它们作为封装对象的属性来创建。在从每个页面都可以链接的 JavaScript 文件中放置以下代码：

```
var GLOBALS = (function() { return this; }());
```

接下来使用 GLOBALS.Xxx，这里的 Xxx 是任何你可能需要的全局变量。这至少可以确保你的全局变量容易辨识。

一个更好的方法是干脆不使用全局名称空间和 this 对象，而改用全新的全局字典对象。将其如下面这样初始化：

```
var GLOBALS = GLOBALS || {};
```

如果在应用程序中使用多个需要引用 GLOBALS 的 JavaScript 文件，那么你就要将上面那行代码放置在每个文件的第一行。该语法中的||会确保不会在每次文件被处理时重新初始化。

注意：

JSLint(可以在 http://www.jslint.com 找到)——用于 JavaScript 代码统计分析的一款在线工具——能够帮助捕获你代码中的反模式，包括 var 关键字的缺失。

4. 变量与提升

提升是 JavaScript 的一个特性，通过它开发人员可以在范围中的任何地方声明变量并在该范围中的任何地方使用这些变量。在 JavaScript 中，可以先使用变量，之后再声明它(作为 var)，如下所示：

```
function() {      // Not allowed in C#
    mode = 1;
    ...
    var mode;
}
```

其整体行为就如该 var 语句被放置在顶部一样。历史上，该特性的引入就是为了 JavaScript 具有尽可能低的使用门槛。不过，当你使用 JavaScript 来编写代码的重要部分时，提升会成为混乱的明显来源并且变得容易出错。一个良好习惯是将你所有的变量放置在每个函数的顶部——最好是放置在单个 var 语句中，如下所示：

```
function() {
   var start = 0,
       total = 10,
       sum = function (x,y) {return x+y;},
       index;
   ...
}
```

JSLint 可被指示来捕获在单个函数中使用的多个 var 语句，并提示你与此模式有关的信息。

5. 对象

不将 JavaScript 归类成面向对象语言的主要原因是，你从 JavaScript 得到的对象定义与被普遍接受的对象概念不同，后者是你从像 C++或 C#这样的经典面向对象语言中得到的。

在 JavaScript 中，对象是一个名称/值对的字典。其对象的设计结构是隐式的，你无法获取。JavaScript 对象通常只具有数据，但可以添加行为。对象的(显式)结构可以在任何时候使用新方法和属性进行修改；但其隐式结构永远无法修改。正是由于该隐式设计结构，JavaScript 中任何明显的空白对象都仍然具有一些属性，比如原型(稍后我将介绍原型)。

请记住，存储对象的变量实际上并不会包含该对象的字典；它们只是引用该对象的属性包而已。属性包与变量存储不同，并且不同的变量可以引用相同的数据包。new 运算符会创建新的属性包。当你传递一个对象时，只需要传递包的引用即可。

将一个成员添加到 JavaScript 对象只会在特定实例起作用。如果想要将一个新成员添加到被创建成某类型的所有实例，就必须将该成员添加到对象的原型。

```
if (typeof Number.prototype.random === "undefined") {
    Number.prototype.random = function() {
        var n = Math.floor(Math.random() * 1000);
        return n;
    };
}
```

注意：

增加原生对象的原型，被一些开发人员认为是不好的做法(或者说，至少值得讨论)，因为它会使代码降低可预测性并损失可维护性。总体而言，我相信这是值得怀疑的，主要是观点角度的问题。我的观点是：注意潜在的问题并按照你感觉舒适的方式来进行处理。

可以使用 Object 类型来创建值和方法的聚合，这是你在 JavaScript 中使用 C#对象最可行的方式。Object 构造函数的直接使用(如下所示)通常会被忽视：

```
var dog = new Object();
dog.name = "Jerry Lee Esposito";
dog.getName = function() {
    return this.name;
}
```

一种更好的方式需要使用对象字面值，如下所示：

```
var dog = {
   name: "Jerry Lee Esposito",
   getName: function() {
      return this.name;
   }
};
```

如果使用 Object 的构造函数，解释器就必须解析构造函数调用的范围。在 JavaScript 中，不能确保范围中存在的本地对象具有与全局对象不同的名称。因此，解释器必须顺着堆栈找出与构造函数所应用的最近的定义。除了这个性能问题之外，直接使用构造函数还不能将对象作为字典传输，而字典恰恰是 JavaScript 编程的关键。

6. 函数

在 JavaScript 中，函数就是捆绑成块的一些代码，并且可选择性为其命名。如果函数没有指定名称，则其就是匿名函数。你要像对待对象一样对待函数：它们可以具有属性，并且可以传递它们并与之进行交互。

在 JavaScript 中，匿名函数是函数编程的支柱。匿名函数是 λ 演算(lambda calculus)[①]的直接分支，或者如果愿意的话，也可以成为一种老式函数指针的语言改造。以下是匿名函数的

① λ 演算(Lambda calculus)是一种由 Alonzo Church 和 Stephen Kleene 于 1930 年开发的程序语言数学基础，它可以用来表示所有可计算的函数。在对可计算性(也是可构造性和有效计算性)概念标准化的努力中，Church 和 Kleene 开发出一种只需要简单的句法和少量的语法约束强力有效的语言。这种语言把函数程序当成它的一个参数(一个函数是一套规则)，把实体像其他变量一样表示，一个函数功能对另一个对的调用，或者当成“lambda 抽象”(用希腊字母 lambda 表示的一个函数定义维抽象运算)。λ 演算和类似合成逻辑和分类系统理论一样，他们是学习数学逻辑和计算机编程语言的重要基础。

一个示例：

```
function(x, y) {
  return x + y;
}
```

常规函数与匿名函数之间的唯一区别就是其名称不同(或者也可以说没有区别)。

使用函数的主要原因有两个：一个是创建自定义对象，另一个当然就是定义可重复行为。这两个原因中，前者在 JavaScript 中最受关注。请思考以下代码：

```
// The this object is implicitly returned
var Dog = function(name) {
  this.name = name;
  this.bark = function() {
     return "bau";
  };
};
```

要使用 Dog 对象，你需要使用经典的 new 构造函数对其实例化，如下所示：

```
var jerry = new Dog("jerry"); // OK
var hassie = Dog("hassie");   // Doesn't throw but, worse, may alter the
                                 application's state
```

麻烦的是，就算你忘记了 new 运算符，也不会得到任何异常且你的代码会继续运行，但是使用的 this 对象现在就变成全局的 this 对象了。这意味着你潜在地改变了应用程序的状态。下面是安全的应对措施：

```
var Dog = function(name) {
   var that = {};
   that.name = name;
   that.bark = function() {
      return "bau";
   };
   return that;
};
```

不同之处在于，现在你显式地创建并返回了一个名为 that 的新对象。这就是俗称“使用 that 而非 this”的模式。

11.1.2　JavaScript 中的面向对象

曾几何时，网页中的 JavaScript 代码仅限于 DOM 基本操作的几行代码。也就没有必要将这些代码设计为可重用块并小心地将其附加到页面元素。尽管 JavaScript 代码的平均复杂

水平与不久以前差不多相同，但每个页面上的代码质量要求已显著提高了。如今需要各种封装形式，因为 JavaScript 代码现在就像是其自身上的小应用程序一般。你需要获得可重用性以及(如果不是特殊情况的话)可维护性。再者，你需要确保你的代码独立运行，因为在 JavaScript 中，如果目前为止没有明显的指示，很容易就会缺失变量、破坏全局性并且输错名称。在这一点上，JSLint 就大有助益了，但它并不像一个编译器。

为了完成有关 JavaScript 语言基础知识的讨论，让我介绍一下闭包和原型——可以用来在 JavaScript 中实现面向对象的两种方式。

1. 让对象具有类的外观

在我开始介绍闭包和原型之前，让我先介绍一下原生 Object 类型及其用途。正如之前所描述的，可以使用 new 关键字来创建一个像字典一样的新对象。接着，你要将数据填充进去并通过为属性名称编写函数来添加一些方法。下面是一个示例：

```
var person = new Object();
person.Name = "Dino";
person.LastName = "Esposito";
person.BirthDate = new Date(1979,10,17);
person.getAge = function() {
  var today = new Date();
  var thisDay = today.getDate();
  var thisMonth = today.getMonth();
  var thisYear = today.getFullYear();
  var age = thisYear-this.BirthDate.getFullYear()-1;
  if (thisMonth > this.BirthDate.getMonth())
      age = age +1;
  else
  if (thisMonth == this.BirthDate.getMonth() &&
      thisDay >= this.BirthDate.getDate())
      age = age +1;
  return age;
}
```

你使用了一个基于人的属性来建模的对象；但你不会真正具有一个 Person 对象。正如你早前看到的，这样做会有可读性和性能方面的问题。此外，这个对象被少有地定义为多行。

闭包和原型提供了另一种方式来定义类型的布局，并且它们是用于在 JavaScript 中进行面向对象编程的原生机制。

2. 使用闭包

闭包是编程语言的一个通用概念。应用到 JavaScript 时，闭包就是一个能在相同上下文

内同时定义变量和方法的函数。以这种方式，最外层(匿名或已命名的)函数会“关闭”表达式。这里有一个用于表示 Person 类型的函数的闭包模型示例：

```
var Person = function(name, lastname, birthdate) {
   this.Name = name;
   this.LastName = lastname;
   this.BirthDate = birthdate;
   this.getAge = function() {
      var today = new Date();
      var thisDay = today.getDate();
      var thisMonth = today.getMonth();
      var thisYear = today.getFullYear();
      var age = thisYear-this.BirthDate.getFullYear()-1;
      if (thisMonth > this.BirthDate.getMonth())
          age = age +1;
      else
         if (thisMonth == this.BirthDate.getMonth() &&
             thisDay >= this.BirthDate.getDate())
             age = age +1;
      return age;
   }
}
```

如你所见，闭包就是伪类的构造函数。在一个闭包模型中，构造函数包含了成员声明，而且成员都是真正封装在类中和对类私有的。此外，成员都是基于实例的，这会提高由类使用的内存。这里有一个如何使用该对象的例子：

```
var p = new Person("Dino", "Esposito", new Date( ... );
alert(p.Name + " is " + p.getAge());
```

闭包模型会提供完全封装，但也仅此而已了。要构成对象，你只能借助于聚合。

3. 使用原型

原型模型需要通过 JavaScript 原型对象来定义类的公共结构。以下代码示例显示了如何重写之前的 Person 类以避免闭包：

```
// Pseudo constructor
var Person = function(name, lastname, birthdate) {
    this.initialize(name, lastname, birthdate);
}

// Members
Person.prototype.initialize = function(name, lastname, birthdate)  {
```

```
    this.Name = name;
    this.LastName = lastname;
    this.BirthDate = birthdate;
}

Person.prototype.getAge = function() {
    var today = new Date();
    var thisDay = today.getDate();
    var thisMonth = today.getMonth();
    var thisYear = today.getFullYear();
    var age = thisYear-this.BirthDate.getFullYear()-1;
    if (thisMonth > this.BirthDate.getMonth())
        age = age +1;
    else
      if (thisMonth == this.BirthDate.getMonth() &&
          thisDay >= this.BirthDate.getDate())
          age = age +1;
    return age;
}
```

在原型模型中，构造函数与成员被清晰分离，并且总是需要一个构造函数。就私有成员而言，你无法使用它们了。将其保持在本地闭包中的 var 关键字，并不会在原型模型中应用。所以，可以为你打算用作属性的内容定义获取器/设置器，但支持字段仍将保持可从外部访问。可以借助于一些内部约定，比如你打算用作私有的成员名称的带有下划线的前缀。然而，这只是一个约定而已，没什么能够阻止开发人员访问类作者认为私有的内容。

借助原型特性，就可以通过简单地将派生对象的原型属性设置成“父”对象的实例来达到继承目的。

```
Developer = function Developer(name, lastname, birthdate) {
   this.initialize(name, lastname, birthdate);
}
Developer.prototype = new Person();
```

请注意，总是需要使用这种方式从任何相关成员函数内部引用原型成员。

在原型模型中，所有实例都能共享成员，因为这些成员都是在共享的原型对象上调用的。以这种方式，就能减少每个实例使用的内存大小，也就能够提供更快速的对象实例化过程。除了语法特点之外，原型模型还使得类的定义远比闭包模型更类似于经典的面向对象模型。

4. 普通自定义对象与类层次结构的对比

在闭包与原型之间做选择还应该将性能考量和浏览器性能作为依据。原型在所有浏览器中都具有较短的加载时间。闭包在某些浏览器中运行极佳(例如 IE)但在另一些中则很糟。

原型为微软智能提示提供了更好的支持，并且用在像微软 Visual Studio 这样的支持该特性的工具中时，原型会提供基于工具的语句补全。使用反射，原型还能帮助你获得类型信息。你不必再创建类型实例来查询类型信息了，这在使用闭包的时候是不可避免的。最后，原型使你能够在调试时轻易地浏览私有类成员。

因此，在处理外观和类一样的 JavaScript 对象时，你就有了两个基本选项了。原型是最常被库设计者选择的选项。另外，在 jQuery 中，原型属性也得到了广泛使用。

话虽如此，如果不得不为 Web 前端编写客户端代码的话，我很可能会使用 jQuery，使用大量匿名函数，并且甚至不会使用自定义对象的层次结构。我肯定会创建自定义对象，但我只会将其用作数据和行为的普通而平滑的容器，其中不具有继承性或多态性。另一方面，如果我不得不编写自己的框架来支持某些服务器端基础架构，我很可能会选择一种更经典的面向对象方法。然而，在那种情况下，我可能会考虑使用一个现成的库而非创建一个自己的库。对此，MooTools(http://mootools.net)就是一个极佳的选择。

11.2　jQuery 的执行摘要

我认为如果现在要在 Web 视图中编写 JavaScript 代码的话，你很可能使用 jQuery。如果没有，那么应该使用；唯一我认为可以不使用 jQuery 的合理场景是移动网站。要能让你在 DOM 操作和事件处理方面省力，jQuery 库肯定不是你能选用的唯一 JavaScript 库。不过，它是全球的业界标准。我认为 jQuery 几乎就是 JavaScript 语言的扩展，并且肯定是任何 Web 开发人员 JavaScript 技能的扩展。

在本章中，你不会看见对 jQuery 的广泛介绍。对于这方面的内容，可以选择一些合适的书籍或查看 http://docs.jquery.com/Main_Page 处的在线文档。如果要寻找更具可读性的在线内容而不是枯燥的文档，请查看 http://jqfundamentals.com/book。

在这一章，我会介绍 jQuery 中关键概念的概况，对这些概念的足够理解能让你快速掌握 jQuery 库中的编程细节和特性。

11.2.1　DOM 查询与包装集

jQuery 库在世界范围内大获成功的主要原因在于其对函数及 DOM 编程的独特混合。该库的工作原理是选择 DOM 元素并将函数应用于这些元素，而这正是客户端 Web 开发人员需要在大多数时候完成的任务。

1. 根对象

jQuery 库的根就是 jQuery 函数。以下是该库的整体结构：

```
(
    function( window, undefined )
    {
        var jQuery = (function() {
            // Define a local copy of jQuery
            var jQuery = function(selector, context) {
                ...
            }
            ...
            return jQuery;
        })();

        /* the rest of the library goes here */
        ...
        window.jQuery = window.$ = jQuery;
    }
) (window);
```

该嵌套 jQuery 函数被映射成浏览器窗口对象的扩展并且用流行的$函数作为别名。该函数具有以下原型：

```
function(selector, context)
```

选择器表示在 DOM 上运行的查询表达式；上下文表示从 DOM 的哪个部分运行该查询。如果没有指定上下文，则 jQuery 函数会在整个页面 DOM 中查找 DOM 元素。

jQuery 函数通常返回一个包装集——也就是 DOM 元素的集合。非常棒的是，这个包装集仍然是 jQuery 对象，可使用相同的语法来查询，并产生一系列查询。

2. 运行查询

库名称中的 query 这个词说明了一切(j 代表的是 JavaScript，这一点你已经知道了)——jQuery 库的主要目的是运行针对 DOM 的(明智)查询并执行针对返回项的操作。

该库背后的查询引擎已经远不止简单的搜索功能了，例如，你能在 DOM 找到的原生的 document.getElementById(以及相关函数)这样一些简单搜索。jQuery 的查询功能使用了强大的 CSS 语法，它为你赋予了惊人的表达性水平。只有在 HTML5 的 DOM 中你才会发现类似的查询表达性，而 HTML5 中的 CSS 语法受到了广泛而统一的支持。

一个查询结果就是一个包装集。包装集是包含 DOM 元素集合的对象。元素会按照其在原始文档中出现的顺序被添加到集合中。

即使没有找到匹配元素，包装集也绝不会为空。可以通过查看 jQuery 对象的 length 属性来检查包装集的实际大小，如下所示：

```
// Queries for all IMG tags in the page
var wrappedSet = new jQuery("img");
var length = wrappedSet.length;
if (length == 0)
alert("No IMG tags found.");
```

请注意上面所示的表达式，通过它你就能得到包装集，该表达式等效于更常用的$(“img”)。

包装集并不是一个特殊的数据容器。“包装集”是一个表明查询结果的特定 jQuery 术语。

3. 枚举包装集的内容

为了循环遍历包装集中的元素，你要使用 each 函数。此函数会将一个函数用作参数并基于每个元素调用它：

```
// Prints out names of all images
$("img").each(function(index) {
   alert(this.src);
});
```

传递给每个元素的回调函数会接收到当前迭代的从 0 开始计数的索引。非常棒的是，你不需要亲自检索相应的 DOM 元素；只需要使用 this 关键字来引用当前正在处理的元素即可。如果回调函数返回 false，迭代就会停止。请注意，each 是一个非常通用的函数，可用于不存在更特定 jQuery 函数的任何任务。如果发现已经存在一个 jQuery 函数能够完成你想要通过编码进行处理的任务，请务必使用原生的函数。

可以使用 length 属性来读取包装集的大小。也可以使用 size 函数，但 length 属性会稍微快一点。

```
// You better use the length property
alert($("img").size());
```

get 函数会从 jQuery 对象中提取包装集并将其作为 DOM 元素的 JavaScript 数组来返回。相反，如果传递一个 index 参数，它将返回在包装集中指定的从 0 开始计数的位置处的 DOM 元素。

```
var firstImage = $("img")[0];
```

请注意，get 函数(以及索引)会中断 jQuery 的连贯性，因为它会返回 DOM 对象或 DOM 对象的数组。你不能进一步将 jQuery 函数应用于 get 调用的结果。

包装集上还有更多可用的操作，其中许多你都可以通过插入方式来添加。

11.2.2　选择器

查询是用选择器来描述的。选择器就是一个表达式，在经过正确的计算后，会选择一个或多个 DOM 元素。在 jQuery 中，你有三种基本类型的选择器：基于 ID 的选择器、层叠样式表(CSS)和标签名称。此外，选择器可以从使用特设运算符联合的多个更简单选择器的组合中产生。在这种情况下，可以使用复合选择器。

1. 基本选择器

ID 选择器会通过 ID 来选取 DOM 元素。ID 选择器通常只会选择一个元素，除非页面中有多个元素共享一个相同的 ID——这一条件违背了 HTML DOM 标准，但这在现实情形中并非不常见。下面是 ID 选择器的语法：

```
// Select all elements in the context whose ID is Button1
$("#Button1")
```

最前面的#符号指示 jQuery 如何解释跟随其后的文本。

基于 CSS 的选择器会选取共享指定 CSS 类的所有元素。其语法如下：

```
// Select all elements in the context styled with the specified CSS class
$(".header")
```

在这个例子中，最前面的圆点(.)符号会引导 jQuery 将后面的文本解释成 CSS 样式名称。

最后，基于标签的选择器会选取带有指定标签的所有元素，比如所有<img>标签、所有<div>标签，或者你指定的任何标签。在下面的例子中，选择器由普通标签名称组成——在最前面不需要符号：

```
// Select all IMG elements in the context
$("img")
```

如前所述，你还可以串联两个或更多个选择器来组成更特定的选择器。

2. 复合选择器

通过多个运算符就能实现串联。例如，空格会选取满足第二个选择器的所有元素并将这些元素作为匹配第一个选择器的元素的子代元素。下面是一个示例：

```
// Select all anchors contained within a DIV
$("div a")
```

上面所示的选择器在功能上等效于以下 jQuery 表达式：

```
$("div").find("a");
```

类似于空格，>运算符会选择由第一个选择器匹配的元素的直接子元素(并不是所有子代元素)：

```
// All anchors direct child elements of a DIV
$("div > a")
```

上面的选择器在功能上等效于以下 jQuery 表达式：

```
$("div").children("a")
```

选择器的普通串联可由逻辑 AND 条件产生。例如，请思考以下查询：

```
$("div.header.highlight")
```

它会选择所有同时使用标头类和高亮类为样式的<div>元素。

+运算符——相邻运算符——会在第一个选择器选择的元素之后紧接着选择第二个选择器中的同级元素。下面是一个示例：

```
// All P immediately preceded by A
$("a + p")
```

~运算符——下一位运算符——类似于+，只不过它选择的是紧随其他元素之后的同级元素。下面是一个示例：

```
// All P preceded by A
$("a ~ p")
```

作为替代，通过使用逗号，就可以返回由复合选择器查询出的元素合集。在运算中，逗号表示逻辑 OR 选择器。以下例子会选取是 A 或 P 的元素：

```
// All A and all P
$("a, p")
```

除了简单运算符，可以使用筛选器。筛选器就是包含一些自定义逻辑以进一步限定所选元素的特定 jQuery 表达式。

3. 预定义筛选器

可以通过在位置、内容、特性和可见性上应用筛选器来进一步改进选择器。筛选器是一种内置函数，被应用于由基本选择器返回的包装集。表 11-1 列出了 jQuery 中的位置筛选器。

表 11-1　位置筛选器

筛　选　器	描　　述
:first	返回匹配上的第一个 DOM 元素
:last	返回匹配上的最后一个 DOM 元素
:not(selector)	返回不匹配指定选择器的所有 DOM 元素
:even	返回在从 0 开始计数的索引中位于偶数位置的所有 DOM 元素
:odd	返回在从 0 开始计数的索引中位于奇数位置的所有 DOM 元素
:eq(index)	返回在包装集中位于指定从 0 开始计数的位置的 DOM 元素
:gt(index)	返回在从 0 开始计数的索引中位置大于指定索引的所有 DOM 元素
:lt(index)	返回在从 0 开始计数的索引中位置小于指定索引的所有 DOM 元素
:header	返回是标头的所有 DOM 元素，比如 H1、H2 等
:animated	返回当前正通过 jQuery 库中某些函数进行处理的所有 DOM 元素

表 11-2 列出的所有筛选器都可以帮助你选出一个父元素的子元素。

表 11-2　子筛选器

筛　选　器	描　　述
:nth-child(expression)	返回匹配指定表达式的任意父元素的所有子元素。该表达式可以是一个索引或数列(例如，3n+1)，包括像奇数和偶数这样的标准数列
:first-child	返回是其父元素的第一个子元素的所有元素
:last-child	返回是其父元素的最后一个子元素的所有元素
:only-child	返回是其父元素的唯一一个子元素的所有元素

一个功能特别强大的筛选器是 nth-child。它支持大量的输入表达式，如下所示：

```
:nth-child(index)
:nth-child(even)
:nth-child(odd)
:nth-child(expression)
```

第一个格式会选择源选择器中所有 HTML 元素的第 n 个子元素。相反，如果指定奇数或偶数筛选器，则会返回位于从 0 开始计数的索引的奇数或偶数位置的所有子元素。

最后，可以给 nth-child 筛选器提供一个数列表达式，比如 3n 表示位置是 3 的倍数的所有元素。以下选择器会选取一个表格中的所有行(标记为 Table1)，这些行位于由数列 3n+1——即 1、4、7 等所确定的位置：

```
#Table1 tr:nth-child(3n+1)
```

表 11-3 列出了用于根据内容筛选元素的表达式。

表 11-3　内容筛选器

筛　选　器	描　　述
:contains(text)	返回包含指定文本的所有元素
:empty	返回没有子元素的所有元素
:has(selector)	返回至少包含一个匹配指定选择器元素的所有元素
:parent	返回至少具有一个子元素的所有元素

就内容筛选器而言，你应该注意 HTML 元素中的任何文本都会被看作是子节点。所以，由空筛选器选择的元素就没有子节点或任何文本。举例：
标签。

一个流行和强大的筛选器类别是特性筛选器。使用特性筛选器，你就可以选出指定特性与某值具有指定关系的 HTML 元素。表 11-4 列出了在 jQuery 中受支持的所有特性筛选器。

表 11-4　特性筛选器

筛　选　器	描　　述
[attribute]	返回具有指定特性的所有元素。该筛选器会选择元素而不管其特性值是什么
[attribute = value]	返回指定特性被设置为指定值的所有元素
[attribute != value]	返回其指定特性具有的值与指定值不同的所有元素
[attribute ^= value]	返回其指定特性具有的内容以指定值开头的所有元素
[attribute $= value]	返回其指定特性具有的内容以指定值结尾的所有元素
[attribute *= value]	返回其指定特性具有的内容包含指定值的所有元素

你还可以通过将两个或更多特性筛选器并排放置来串联特性筛选器，如以下示例所示：

```
var elems = $("td[align=right][valign=top]");
```

该返回集包括所有水平对齐靠右及垂直对齐居顶部的<td>元素。

下面的表达式要复杂得多，揭示了 jQuery 选择器强大的力量及灵活性，因为该表达式组合了不少选择器：

```
#Table1 tr:nth-child(3n+1):has(td[align=right]) td:odd
```

其含义如下：

在元素 Table1 的正文内，选择所有位于 1、4、7 等位置的<tr>元素。接着，只保留<td>元素存在特性 align

等于 right 值的表格行。此外，剩下的行中，只保留索引为奇数的列上的单元格。

其结果是由<td>元素组成的包装集。

最后，还有几个筛选器是与元素可见性有关的。:visible 筛选器会返回当前可见的所有元素。:hidden 筛选器会返回当前从视图中隐藏的所有元素。该包装集还包括类型特性等于"hidden"的所有输入元素。

4. filter 与 find 的对比

要进一步限定一个查询，可以在包装集上使用 find 或 filter 函数。当然，它们并不相同。

filter 函数会探查当前的包装集以匹配元素，绝不会检查 DOM 以获取子代元素。相反，find 函数会对包装集内部的每个元素进行检查以获得匹配表达式的元素。不过，这样的话，该函数会探查包装集中每个元素的 DOM。

5. 包装集上的串联操作

jQuery 库提供了各式各样的函数，可以将它们应用到包装集内容上(要获得一个完整列表，只能借助于在线文档，或找到一本深入讨论这方面内容的书)。可以将函数调用串联起来，因为由查询返回的任何包装集反过来会作为另一个 jQuery 对象，以便进一步查询。例如，以下表达式就能正常运行：

```
$(selector).hide().addClass("hiddenElement");
```

上述表达式首先会对视图隐藏所有匹配的元素，然后为每一个元素添加一个指定 CSS 类。

可以将操作归类成能够在包装集上执行的几个分组，如表 11-5 中所描述的。

表 11-5　包装集上的操作

作　用	描　述
DOM 操作	创建 DOM 树、添加/移除元素，或修改现有元素
绑定事件	将处理程序绑定和解除绑定到由 DOM 元素发起的事件
样式化	为所选元素应用、移除或切换 CSS 类并得到或设置单个 CSS 属性
可见性	使用过渡效果(例如渐变)及持续时间显示和隐藏 DOM 元素

此外，在 jQuery 中你会发现其他两个实用分组——缓存和 Ajax 调用——它们会处理包装集的内容，不过不能将其严格地看作是包装集上可用的操作。

11.2.3　事件

处理事件在 JavaScript 编程中是常见的活动。jQuery 库提供了大量函数将处理程序绑定

到以及解除绑定到由 DOM 元素发起的事件。

1. 绑定和解除绑定

bind 和 unbind 这对函数用于将回调函数附加到指定事件。这里有一个示例，其中匹配选择器的所有元素都将为 click 事件附加相同的处理程序：

```
$(selector).bind("click", function() {
...
});
```

你要使用 unbind 函数来将当前已定义的处理程序与指定事件分离：

```
$(selector).unbind("click");
```

请注意，unbind 函数并不会移除那些通过 onXXX 这样的特性直接在标记中插入的处理程序。

jQuery 库还定义了不少直接函数来绑定指定事件。已有的用于事件的配套函数包括 click、change、blur、focus、dblclick、keyup 等。以下代码显示了如何将处理程序绑定到 click 事件：

```
$(selector).click(function() {
...
});
```

进行不带回调的调用，相同的事件函数会产生调用当前处理程序的效果，如果该处理程序被注册过的话。例如，以下代码模拟了用户单击指定按钮的情况：

```
$("#Button1").click();
```

可以通过使用 trigger 函数以一种更通用的方式来达到相同效果：

```
$("#Button1").trigger("click");
```

事件处理程序会接收一个 jQuery 内部对象——Event 对象。该对象会提供一个用于事件的统一编程接口，该接口与万维网联盟(W3C)的倡议密切相关，并且它解决了某些浏览器造成的轻微不同实现所引起的差异性问题：

```
$("#Button1").click(function(evt) {
    // Access information about the event
    ...

    // Return false if you intend to stop propagation
    return false;
});
```

Event 对象的属性特征包括鼠标坐标、事件的 JavaScript 时间、使用的是哪个鼠标按钮，以及事件的目标元素等。

2. 动态事件绑定

动态绑定是 jQuery 的一个很棒的特性，通过它可以在整个页面生命周期中保持对 DOM 元素指定子集的事件绑定跟踪。换句话说，如果选择动态绑定而非普通绑定，就确保了任何动态添加的能匹配选择器的元素都会自动附加相同的处理程序。从 jQuery1.7 开始，你就该通过 on 和 off 函数来操作动态绑定了。下面是一个示例：

```
$(document).on("click", ".specialButton ", function() {
...
})
```

所有用 specialButton CSS 样式修饰的按钮都作为 click 事件的处理程序附加的指定函数。要停止某些元素的动态绑定，可以使用 off 函数：

```
$(".specialButton").off("click");
```

使用 on 和 bind(或指定的事件函数，比如 click)之间的区别在于，使用 on 函数时，任何添加到页面的以及用 specialButton 样式修饰的新的 DOM 元素都会自动添加处理程序。但使用 bind 的话就不是这样了。

注意：
如果使用的 jQuery 版本比 1.7 低，就要使用 live 和 die 方法作为 on 和 off 的替代。其语法稍有不同，因为 live 方法不会将选择器用作参数。相反它会直接应用于选择器。

3. 准备页面和 DOM

在客户端开发初期，只有一个地方你能够放置网页的初始化代码：在窗口对象或<body>标签上的 onload 事件中。页面一完成加载，onload 事件就会触发——也就是说，在所有链接图片、CSS 样式以及脚本完成下载之后。不过，这不能确保此时 DOM 已经被完全初始化并且准备好接受指令。

DOM 中的 document 根对象公开了一个只读的 readyState 属性，这就是为了让你获知 DOM 的当前状态并获悉何时你的页面准备就绪，能够开始执行脚本。使用 readyState 属性是一种绝对可行的方式，但它有一点麻烦。为此，jQuery 提供了其自己的 ready 事件，提示你什么时候可以开始在框架中进行安全调用了。

```
<script type="text/javascript">
$(document).ready(
```

```
  function() {
    alert("I'm ready!");
  });
</script>
```

可以在页面或视图中拥有多个 ready 调用。指定多个 ready 调用时，jQuery 会将指定函数推送到内部堆栈并在 DOM 实际准备好之后按顺序对其进行处理。

ready 事件仅会在文档级别触发；你不能为单个元素或包装集中的任何元素集合定义该事件。

注意：

onload 事件是在 HTML 和所有辅助资源都加载之后调用的。ready 事件是在 DOM 初始化之后调用的。这两个事件的运行顺序可以随意安排。onload 事件不能确保页面 DOM 已加载；ready 事件不能确保所有资源(比如图片)已经加载。

11.3　JavaScript 编程特性

如今，你常常将 JavaScript 用于某些客户端逻辑和输入验证。可以使用 JavaScript 来下载远程服务器的数据，实现类似 Windows 的效果，如拖放，用于缩放、用于模板、用于弹出和图表效果、用于本地数据缓存，以及用于管理围绕页面的历史记录和事件。JavaScript 被用于大的代码块，这些代码块具有较高水平的可重用性，且彼此之间需要安全地隔离。

换句话说，你希望你的 JavaScript 代码具备可维护性并且无侵入性。

11.3.1　无侵入性代码

多年来，编写带有能够显式附加到 JavaScript 事件处理程序的客户端按钮的 HTML 页面非常普遍。下面是一个典型示例：

```
<input type="button" value="Click me" onclick="handleClick()" />
```

从纯粹功能的角度看，此代码没什么问题；它会如预期般运行，当用户单击按钮时运行 handleClick 这个 JavaScript 函数。当 JavaScript 仅用于网页的辅助功能时，在很大程度上可以接受这种方式；不过，当有大量 JavaScript 代码表示页面或视图的重要部分时，就难以处理了。

1. 使用代码来设计视图样式

“无侵入性 JavaScript”这一表述最近很流行，它是指最好不要在 HTML 元素与 JavaScript 代码之间使用显式链接。从某种程度上说，无侵入性 JavaScript 就是 CSS 类的对应脚本。

使用 CSS，可以编写普通的 HTML 而无需内联的样式信息，还可以通过使用 CSS 类将

样式添加到元素。同样的，当 DOM 就绪时，可以避免使用事件处理程序特性(onclick、onchange、onblur 等)，并使用单个 JavaScript 函数来附加处理程序。下面是一个简明但有效的无侵入性 JavaScript 的示例：

```
<script type="text/javascript">
    $(document).ready(function () {
        $("#Button1").bind("click", function () {
            var date = new Date().toDateString();
            alert(date);
        });
    });
</script>

<h2>JavaScript Patterns</h2>
<fieldset>
   <legend>#1 :: Click</legend>
   <input type="button" id="Button1" value="Today" />
</fieldset>
```

可以将整个<script>块移动到一个单独的 JavaScript 文件并保持你视图的整洁与可读性。

2. 无侵入性 JavaScript 的实用规则

无侵入性 JavaScript 建立了一个基础原则——任何网页中的任何行为都必须是可注入的依赖项并且不是一个构建块。富 JavaScript 代码是由处理逻辑和管理用户界面(UI)的代码组成的。UI 逻辑需要知道 DOM 和视图的结构。这必然会创建一个依赖性。可以接受这样的依赖性，但如果绕过依赖性就更好了。

限制 UI 依赖性影响的一个方法是使用模板和像 KnockoutJS 这样的特设库来呈现页面。可以在 http://knockoutjs.com 获得更多有关 KnockoutJS 的信息。

11.3.2　可重用封装和依赖性

越来越多的页面是广泛地基于 JavaScript 的，造成了越来越多的页面结构组件化的问题。我们来探讨一种被广泛认可的在 JavaScript 中封装代码的方式，这些代码对外部库没有依赖性。

1. 名称空间模式

JavaScript 编程的指导原则是将相关属性——包括全局属性——分组到容器中。当多个脚本文件之间共享这样的容器时，可能就很难决定(以及强制)哪一个文件承担初始化容器和子对象的职责。下面是一个可用于定义全局容器的简单语法的示例。

```
var GLOBALS = GLOBALS || {};
```

这一技巧是可行的，但当你有几个嵌套对象需要管理时，用起来通常会很麻烦。这时名称空间模式就能派上用场了。

名称空间模式由一段代码构成，它针对以圆点分开的字符串的令牌(例如 C#或 Java 名称空间)进行迭代并确保初始化对象的正确层次结构。namespace 函数会确保代码不会被破坏并跳过已有的实例。这里有一些示例代码：

```
var GLOBALS = GLOBALS || {};

GLOBALS.namespace = function (ns) {
   var objects = ns.split("."),
         parent = GLOBALS,
         startIndex = 0,
         i;

   // You have one GLOBALS object per app. This object already exists if you
   // can call this function. So you can safely ignore the root of the namespace
   // if it matches the parent string.
   if (objects[0] === "GLOBALS")
      startIndex = 1;

   // Create missing objects in the namespace string
   for (i = startIndex; i < objects.length; i++) {
      var name = objects[startIndex];
      if (typeof parent[name] === "undefined")
         parent[name] = {};
      parent = parent[name];
   }
   return parent;
};
```

引用 namespace 函数之后，随后你就能放置以下调用：

```
GLOBALS.namespace("Widgets");          // GLOBALS has a Widgets property
GLOBALS.namespace("GLOBALS.Widgets");  // GLOBALS has a Widgets property
```

这两个调用是等效的，并且都能保证 GLOBALS 具有一个初始化的 Widgets 属性。思考以下代码：

```
// GLOBALS has a Widgets property, and Widgets has a NewsBox property
GLOBALS.namespace("GLOBALS.Widgets.NewsBox");
```

namespace 函数会继续迭代并同时确保 Widgets 具有 NewsBox 属性。

重要提示：

尽管此处所示的实现可以被认为是比较标准的，但我觉得有必要为 Stoyan Stefanov 就这段代码以及模块模式一节给予我的启发表示感谢。Stoyan 是 *JavaScript Patterns*(O'Reilly Media，2010)这一优秀书籍的作者。我建议想要进一步详细了解与本章内容有关的知识的人都去读读这本书。

2. 模块模式

模块模式提供了一种封装自包含代码块的方法，借助该方法可以从项目中轻易地添加或移除该代码块。这个模式将一个经典的 JavaScript 函数封装成了一个立即执行函数，它能保障数据的私密性，并确保只有你显式地展示为公共的内容才会作为公共内容被客户端实际感知。在 JavaScript 中，立即执行函数是内联定义并将立即执行的函数。其语法如下：

```
 (
   function(...) {
     // Body
   } (...)
);
```

让我们使用模块模式来构建一个从指定源抓取新闻的自包含部件。假设你将下面所有的代码都保存到一个 JavaScript 文件：

```
GLOBALS.namespace("Widgets.News");
GLOBALS.Widgets.News = function () {
    var localUrl = "...",
        localWidget = "",
        localBuildWidget = function (items) {
            var numOfNews = items.length;
            if (localSettings.maxNews >0)
                 numOfNews = Math.min(localSettings.maxNews, numOfNews);

            var buffer = "<table rules='rows'>";
            for (var i = 0; i < numOfNews; i++) {
                buffer += "<tr><td>" + items[i].Title + "</td></tr>";
            }
            buffer += "</table>";
            localWidget = buffer;
        },
        localSettings = {
            maxNews: 5,
            autoRefreshEvery: 0
        };
```

```
        return {
            load: function (selector, settings) {
                if (settings != null)
                    localSettings = settings;

                $.getJSON(localUrl)
                    .done(function (data) {
                        localBuildWidget(data);
                        $(selector).html(localWidget);
                    });
            },
            getHtml: function () {
                return localWidget;
            }
        };
    } ();
```

该代码定义了一个逻辑块，其中包含下载和对来自源的大量新闻进行格式化所需的所有逻辑。

使用模块模式，你要返回一个对象将你想要使其可见的公共 API 展示出来。下面的代码揭示了你如何使用一个模块：

```
<script type="text/javascript" src="@Url.Content
                        ("~/content/scripts/module-news.js")"></script>
<script type="text/javascript">
    GLOBALS.Widgets.News.load("#twitter-box", {maxNews: 10});
</script>
```

首先链接包含部件的单个文件，然后传递合适的设置对其初始化。这个示例中的部件得到了 UI 的 jQuery 选择器，该部件会在 UI 上进行图形化变更——具体来说，该部件会在 UI 上插入带有所选新闻列表的 HTML 表格。

对于模块模式的实现，名称空间模式并非是必须的，但使用名称空间模式能起到很大的帮助。

11.3.3　加载脚本和资源

网页中的脚本越来越多地意味着有越来越多的脚本文件要下载。这不久可能会变成一个严重的问题并需要进行处理。当页面有几个脚本的时候，浏览器可以操作的并行处理能力就会显著降低。这样一来页面的加载时间也就相应变大了。我们来看看个中缘由。

1. 下载总是同步进行的

HTTP/1.1 规范建议浏览器不要同时从每个主机名下载两个以上的组件。然而，这对于脚

本文件是绝对行不通的：浏览器总是同步下载脚本文件，并且一次只下载一个。其结果就是，总的下载时长至少等于下载单个文件所需的时长之和，且更糟的可能是，下载一个脚本文件时浏览器处于空闲状态。页面呈现只有在脚本文件下载、解析并执行完成后才会继续进行。

浏览器实现同步下载的主要原因是保障安全性。实际上，脚本文件总是可能会包括有像 JavaScript 立即执行函数或 document.write 这样的会修改当前 DOM 状态的指令。

2. 底部脚本

要改善页面加载性能，可以使用一个简单的技巧，也就是将脚本文件的所有链接移动到页面的底部，放在</body>标签之前。当你这样处理的时候，浏览器就不需要中断页面呈现过程来加载脚本了。然后浏览器就可以尽可能快地提前显示页面视图了。

尽管手动将脚本放置到底部是最安全的方式，但有另一个选项让你以声明形式进行设置并无须借助编写或导入特设的 JavaScript 代码：这就是 defer 特性。

```
<script src="..." defer="defer"></script>
```

defer 特性是在 HTML 4 规范中引入的，它会指示浏览器是否能将脚本加载延迟到页面处理结束时进行。用 defer 特性修饰的<script>标签会隐式表明，该脚本不会进行任何直接的文档写操作，并且将其放在结束时加载是安全的。defer 特性的作用类似于 HTML5 规范中的 async 特性。

3. 处理静态文件

Web 开发的指导原则规定，在你消除了像脚本、样式表以及图片这样的静态文件对性能产生的不良影响之后，差不多就完成了优化工作。有两种主要方式来最小化静态资源的下载时长，且两者并不相互排斥。

最显而易见的技巧是减小要下载的文件的大小。第二个最明显的技巧是完全不要下载它们。在详细介绍之前，我事先声明，这些优化技巧应该在你完成开发之后进行。如果在开发过程中应用这些技巧，则大多数技巧只会给你带来挫折感并拖累你的进度。

减小可下载文件的大小意味着压缩其内容。浏览器会通过 Accept-Encoding 标头通知 Web 服务器它们所支持的压缩类型。第 8 章“自定义 ASP.NET MVC 控制器”揭示出，在 ASP.NET MVC 中可以将合适的响应标头添加到任何控制器操作，并且指示 Web 服务器为该响应使用受支持的编码。还可以在 Web 服务器级别为静态资源直接开启压缩。值得注意的是，你不应对那些已经自行压缩过的文件使用 GZIP 压缩(或者 deflate 压缩)。例如，你不应该对 JPEG 图片进行 GZIP 压缩。尽管 GZIP 压缩能缩减约 50%的大小，但它对已经压缩过的资源作用不大，并且会浪费更多的 CPU 功耗。

第二个要考虑的特性是浏览器缓存。静态资源之所以被称为静态，就是因为它们不会频

繁变更。所以，你为何还要对其反复下载呢？通过为你的静态资源分配一个很长的存续期(通过 Expires 响应标头)，你就能避免浏览器频繁下载这些资源。同样地，也可以在 Web 服务器级别为静态资源进行存续期分配，并通过 Response 对象以编程方式为动态处理的资源进行存续期分配。

在这方面，内容分发网络(CDN)是有用的，因为它能提高浏览器缓存已经包含一个资源的似然性(likelihood)，这是由于该资源可能已经由使用相同 CDN 的其他站点使用相同 URL 进行了引用。请注意，放置一个只有一个应用程序使用的 CDN 文件，不会让你受益太多。

注意：

还有另一个选项来抵消静态资源的开销，就是将那些资源放置在另一台被优化过以返回静态内容的服务器上。或者，你也许可以使用像 Varnish(http://www.varnish-cache.org)这样的反向代理工具来从各种服务器上收集图片，并且将其当作从该代理处原生的资源来返回。通过在 CDN 或网站之前使用反向代理，你就能在降低静态文件的请求方面得到显著收益。

4. 使用图片拼合

为了改进图片的提供服务，可以考虑使用图片拼合。图片拼合就是由多个图片组合而产生的单个图片。构成图片会在形成新图片的过程中并排保存，如图 11-1 所示。



图 11-1　用作图片拼合的组合图片

在页面中，你要使用<img>标签来引用整体图片，并通过使用特定 CSS 样式来选择该图片中你想使用的部分。例如，下面是你需要用来为图 11-1 所示的分段呈现可单击按钮的 CSS：

```
.signup-link
{
    width: 175px;
    height: 56px;
    background-image: url(/images/loginsprite.png);
    background-position: -0px 0px;
    background-repeat: no-repeat;
}

.signup-link:hover
{
    background-position: -177px 0px;
}

.signup-link:active
```

```
{
    background-position: -354px 0px;
}
```

background-position 表示呈现该图片的起始相对位置。例如，当鼠标悬停在该图片上时，浏览器会跳过起始位置 177 个像素再开始呈现图片。这意味着第一个按钮不会显示，并且图片拼合呈现的部分正好就是高亮按钮(正如你在图 11-1 中所看见的，为清晰可见，我添加了 1 个像素的空白线，这就是在计算位置时你必须跳过一个像素的原因)。其实际效果是，你只会具有一个图片、只有一次下载、一个缓存图片以及对用户显示的酷炫效果(参见图 11-2)。

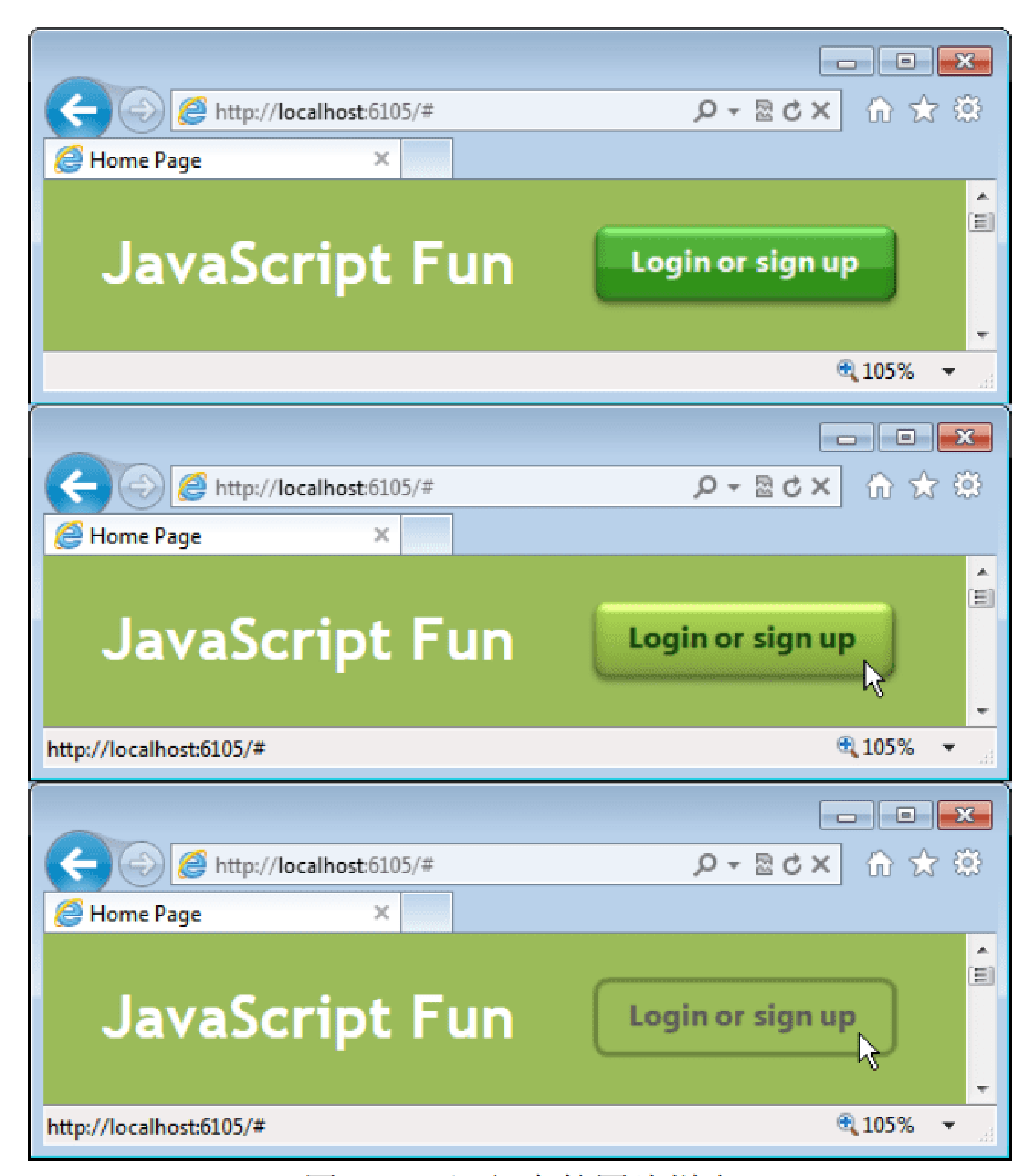


图 11-2　运行中的图片拼合

11.3.4　捆绑和缩小

随着网页持续提供不断丰富的可视化内容，下载像 CSS、脚本和图片这样的相关资源的开销就会显著增加。可以肯定的是，在很大程度上，这些资源都会由浏览器在本地缓存；然而初始的占用空间就真的很难维持了。对于脚本和 CSS 文件来说，GZIP 压缩可以与捆绑和缩小相结合。

捆绑就是将若干不同资源捆在一起成为单个可下载资源的过程。例如，一个捆绑可能由

多个 JavaScript 或 CSS 文件构成。缩小是应用到单个资源的转换。具体来说，缩小就是以某种不更改预期功能性的方式从基于文本的资源中移除所有不必要的字符。这意味着移除注释、空白字符以及新行；一般来说，通常添加的所有字符都是为了具备可读性，但它们会占据空间并且不会真的用于实现功能性目的。

可以同时应用捆绑和缩小，但它们仍然是独立的处理过程。根据需求不同，可以决定只创建捆绑或只缩小个别文件。不过，通常在生产站点上没有理由不去捆绑和缩小所有的 CSS 和 JavaScript 文件。但是在调试时，情况就完全不同了：一个缩小或捆绑的资源很难阅读和逐步调试，所以你绝不会想在调试时启用捆绑和缩小。

市面上存在大量的框架提供捆绑和缩小服务，这些服务具有轻微不同的可扩展性级别和不同的特征集。我相信，在很大程度上，它们都提供了相同的功能；所以，选择哪个框架就只是个人喜好问题而已。如果正在编写一个 ASP.NET MVC 应用程序，捆绑和缩小的原生选项就是微软 ASP.NET Web 优化框架，它是通过一个 NuGet 包提供的。

1. 捆绑相关资源

通常，需要在 global.asax 中以编程方式创建捆绑。按照 ASP.NET MVC 约定，你要在 App_Start 文件夹中创建一个 BundleConfig 类，并从中公开一个静态初始化方法，如下所示：

```
BundleConfig.RegisterBundles(BundleTable.Bundles);
```

一个捆绑仅仅就是一个文件(通常是样式表或脚本文件)集合而已。下面是你需要用来将两个 CSS 文件组合成单个下载的代码：

```
public class BundleConfig
{
   public static void RegisterBundles(BundleCollection bundles)
   {
      bundles.Add(new Bundle("~/all-css").Include(
             "~/content/styles/site1.css",
             "~/content/styles/site2.css"));

      BundleTable.EnableOptimizations = true;
   }
}
```

你创建了一个新的 Bundle 类，并且将用于从视图内部引用捆绑的虚拟路径传递给构造函数。为了把 CSS 文件关联到捆绑，你要使用 Include 方法。该方法采用了一个代表虚拟路径的字符串数组。可以显式指明 CSS 文件，正如上面的例子，或者也可以指明一个模式字符串，如下所示：

```
bundles.Add(new Bundle("~/all-css").Include("~/content/styles/*.css");
```

捆绑是优化的一个形式；因此，它主要用于生产环境中的站点。EnableOptimization 属性是在生产环境中将捆绑设置为启用的便捷方式。请注意，在捆绑被显式开启前，它都不会生效。

2. 捆绑脚本文件

捆绑类在处理 CSS 或 JavaScript 文件方面并没有什么不同。不过，BundleCollection 类有两个特性对于捆绑脚本文件特别有用：顺序和忽略列表。

BundleCollection 类具有一个 IBundleOrderer 类型的名为 Orderer 的属性。正如其名称所表明的，排序器就是一个负责确定实际顺序的组件，你要在其中指定用于下载的被捆绑的文件顺序。默认的排序器是 DefaultBundleOrderer 类。这个类会按照通过 FileSetOrderList 属性设置的顺序捆绑文件，FileSetOrderList 是 BundleCollection 的另一个属性。FileSetOrderList 属性被设计为一个 BundleFileSetOrdering 类的集合。这些类中的每一个都会为文件定义一个模式(例如 jquery-*)，并且 BundleFileSetOrdering 实例的顺序会确定捆绑中文件的实际顺序。例如，基于默认配置，所有的 jQuery 文件就总是会在 Modernizr 文件之前被捆绑。文件的常见分组(比如 jQuery、jQuery UI 和 Modernizr)的顺序是预定义好的；可以随意以编程方式重置和更新顺序。

注意：

DefaultBundleOrderer 类对于 CSS 文件的影响较为有限但并非没有。如果在你的网站中有 reset.css 和/或 normalize.css 文件，这些文件就会在所有其他 CSS 文件前自动捆绑，而且 reset.css 总是位于 normalize.css 之前。使用重置(reset)/标准(normalize)样式表的目的在于，为所有 HTML(重置)和 HTML5(标准)元素提供样式特性的标准集，以便你的页面不会继承特定于浏览器的设置，比如字体、大小、边距。尽管有些推荐内容同时存在于这两个 CSS 文件中，但实际内容取决于你。如果在你的项目中存在具有这些名称的文件，则 ASP.NET MVC 就会尽力确保它们在其他文件之前捆绑。

如果想要重写默认的排序器和忽略预定义的捆绑文件集顺序，有两种选择。第一，可以创建自己的排序器，它会基于每个捆绑运行。下面是一个忽略预定义顺序的示例：

```
public class SimpleOrderer : IBundleOrderer
{
   public IEnumerable<FileInfo> OrderFiles(
         BundleContext context, IEnumerable<FileInfo> files)
   {
      return files;
   }
```

```
}
```

要像如下所示这样使用它：

```
var bundle = new Bundle("~/all-css");
bundle.Orderer = new SimpleOrderer();
```

此外，可以通过使用以下代码来重置所有顺序：

```
bundles.ResetAll();
```

在这个例子中，使用默认排序器或之前所示的简单排序器的效果是相同的。但是，需要注意的是，ResetAll 还会重置所有当前脚本的顺序。

第二个值得注意的特性是忽略列表。它是通过 BundleCollection 类的 IgnoreList 属性来定义的，忽略列表定义了用于匹配文件字符串的模式，这些文件是所选的要包含在捆绑中但却应该被忽略的。忽略列表的主要好处是，可以在捆绑中指定*.js，但可以使用忽略列表来跳过，比如说*.vsdoc.js 文件。IgnoreList 的默认配置顾及到了大多数常见情况(包括*.vsdoc.js 文件)，同时又让你能够进行自定义。

3. 添加缩小

Bundle 类的重心只是将多个资源打包到一起，以便它们能被单次下载捕获和缓存。不过，出于可读性目的，样式表和脚本文件都被填充有空格和新行字符。可读性对人们(以及在调试时)很重要，但这对于浏览器从来就不是问题。下面的字符串是一个能被浏览器完美接受的 CSS 缩小版本示例。如你所见，它不包含任何额外字符。

```
html,body{font-family:'segoe ui';font-size:1.5em;}
   html,body{background-color:#111;color:#48d1cc}
```

你要如何将缩小添加到 CSS 和 JavaScript 文件呢？只要将 Bundle 类修改成 StyleBundle 或 ScriptBundle 类就行了。这两个类都惊人的简单：它们继承自 Bundle，并且仅由一个不同的构造函数构成。

```
public ScriptBundle(string virtualPath)
        : base(virtualPath, new IBundleTransform[] { new JsMinify() })
{
}
```

Bundle 类具有一个接收 IBundleTransform 对象列表的构造函数。这些转换会一个接一个地应用到内容。ScriptBundle 类仅添加 JsMinify 转换器。而 StyleBundle 类添加 CssMinify 转换器。CssMinify 和 JsMinify 都是 ASP.NET MVC 4 的默认缩小器，并且都是基于 WebGrease 框架的。无须多说，如果想要切换到一个不同的缩小器，你需要做的就是创建它对应的类——

IBundleTransform 的实现——并通过构造函数传递它即可。

11.4　本章小结

人们喜欢通过 Web 使用交互式应用程序。出于各种原因，编写这些应用程序的最常用方式仍然是 JavaScript。作为生命力顽强的语言，JavaScript 在 Adobe Flash 和微软 Silverlight 出现之后仍幸运地存活了下来。尽管 Flash 和 Silverlight 仍用于一些 Web 应用程序中，但目前它们已经没有机会吃掉 JavaScript 了。

最初引入 JavaScript 是为了赋予 Web 作者在 HTML 页面中集成某些简单逻辑和操作的能力。JavaScript 的设计目的并非是成为前沿的编程语言。JavaScript 的设计受到许多语言的影响，但其主要因素是简易性。它需要额外的工具来支持超出修改 DOM 元素特性的开发，这也就是像业界标准 jQuery 以及 KnockoutJS 和 AngularJS 所适用的地方。

在第 12 章“让网站对移动端友好”中，我们要进入一种客户端的、密集 JavaScript 应用程序类型的开发：即单页面应用程序。

第 12 章

让网站对移动端友好

不要走在我前面，因为我可能不会跟随。不要走在我后面，因为我可能不会引路。请走在我旁边，做我的朋友。

——*A. Camus*

说起软件，移动端一词常常与特定平台——如苹果公司的 iOS 平台、Windows Phone 或 Android 平台——的本地应用程序联系在一起。这里的共识实际上是你应该让应用程序适用于某些平台，并确保可以在智能手机和(迷你)平板电脑上舒适地浏览网站。最新的设备可以毫无问题地显示几乎所有的网站。这导致许多高管把移动端看成是处理一些本地应用程序的问题，而完全忽视了移动互联网的微妙之处。

我的看法并非如此。我不讨论是否使用(或不使用)移动应用程序，因为该问题太过特定于业务，不便笼统探讨。但是，我确实认为提供一个对移动端友好的网站的确非常重要。更确切地说，我认为，尽管任何一家公司都可以满足于有一个可供显示且能够在智能手机上浏览的网站，但是提供了移动端优化设计的网站用户体验(UX)比仅仅捏拉缩放来填写表单和阅读资讯要好太多了。

在这一章，你将首先回顾一些有助于网站对移动端友好的技术。这些技术中当然包括 HTML5，还进一步包括响应式网页设计(RWD)，并涉及一些特定的 JavaScript 框架，如 jQueryMobile 等。接着，你将了解在付诸实践时你所面临的具体挑战。你会看到如何把两个不同的站点(桌面和移动端的)联系在一起，以便它们在用户眼里显示为同一个网站。第 13 章“构建用于多种设备的站点”会将该讨论深入下去，并解决多设备网站设计的所有重点问题。

12.1 在站点上启用移动端技术

移动端用户在 UX 方面有着较高的期望值；他们希望应用程序提供完整的用户界面(UI)，类似于流行设备(比如 iPhone)的那种，它能基于触摸并且由常见小部件填充，比如选择列表和 iPhone 风格的切换开关。这些小部件并不作为 HTML 的本地元素存在(尚未如此，但预计

很快将出现)，且必须通过每一次使用服务器端控件输出混合的 JavaScript 和标记来模拟。

其底线是，创建一个能有效利用 HTML、层叠样式表(CSS)和 JavaScript 的普通网站是一回事；但要创建一个看起来像或最起码表现得像本地应用程序的引人注目的移动网站却是另一回事了。

一般说来，移动浏览器都会提供对 HTML5 元素的良好支持。这意味着，至少对属于“智能手机”或“平板电脑”的设备来说，可以默认使用 HTML5 元素而不用担心替代方法和嵌入层。因此，让我们总结一下 HTML5 的重要因素吧。

12.1.1　HTML5 对忙碌的开发人员意味着什么

HTML5 标志着 Web 第三个时代的来临，它使 HTML 稳步发展成真正的、完全成熟的应用程序交付格式。HTML5 并不局限于呈现；更准确地说，它还为 Web 以及其他类型的应用程序提供了一系列的新功能。最大的变化是，HTML5 是关于客户端编程和构建可以在浏览器内部运行并与后端进行有限交互(或不交互)的应用程序。

相比其前身(在十多年前定义的)，HTML5 是一个明显更丰富的标记语言。可以说这些年的开发经验并没有白费，因为现在 HTML5 在标准语法中集成了开发人员和设计师在数千个网站中所采用的很多常见做法。在此过程中，HTML5 提出了如何构造 HTML 元素的具体规则，并反对使用过去引入的支持样式元素的标签。新的建议是，应该将 CSS 用于样式元素，并使用特定的(新)标签来定义文档的结构。

1. 语义标记

大多数网站都共同使用一种常见布局，包括页眉、页脚和页面左侧的导航栏。这些效果往往是通过使用向左或向右对齐样式的<div>元素样式来实现的。大多数页面最终使用的都是如下的模板，我们在这本书的所有示例中使用的也是这个模板：

```
<div id="page">
  <div id="header">
    ...
  </div>
  <div id="navbar">
    <ul>
      <li> ... </li>
      <li> ... </li>
      <li> ... </li>
    </ul>
  </div>
    <div id="container">
      <div id="left-sidebar">
        ...
```

```
            <ul>
              <li> ... </li>
              <li> ... </li>
              <li> ... </li>
            </ul>
          </div>
          <div id="content">
            ...
          </div>
          <div id="right-sidebar">
            ...
          </div>
      </div>
      <div id="footer">
          ...
      </div>
    </div>
```

这一模板包含页眉、导航栏、页脚以及它们之间的三列布局。然而，单独使用上面的标记并不会产生预期的效果。为此，你需要在单个<div>元素中添加特设的 CSS 样式，使这些元素处于悬浮状态并保持固定于左边缘或右边缘。

2. HTML5 的特别之处

首先，使用 HTML5 并不意味着你不能使用 CSS 将布局转换成更美观的页面。你仍需要使用差不多相同的 CSS 使页面看起来更吸引人，并将对应部分放置在它们所属的地方。不过，你现在可以用更整洁的方式描述页面，同时对使用 CSS 样式表的设计师来说也更容易阅读。从根本上说，在 HTML5 中可以将普通的<div>元素替换为语义上更有意义的元素，如<header>、<footer>和<article>。下面显示了如何通过使用最新的 HTML 标记来重写前面的模板。

```
    <header> ... </header>
    <nav> ... </nav>
    <article>
      <aside>
        ...
      </aside>
      <section> ... </section>
      <section> ... </section>
      <section> ... </section>
      <aside>
        ...
      </aside>
    </article>
```

```
<footer> ... </footer>
```

<nav>元素会对将在导航栏中出现的链接进行逻辑分组。<article>元素代表了页面任意内容的容器，并集成了<aside>元素和<section>元素。

这些元素都是块元素，它们必须被适当设计以形成符合要求的页面。其他新元素补全了增效元素的列表，如<figure>和<details>。<figure>元素被设计为包含带有说明的图片，而<details>替代了开发人员用来隐藏可选内容和通过 JavaScript 显示该内容的标准隐藏<div>。

3. 本地可折叠元素

新的<details>元素在功能上等同于<div>，但其内在内容由浏览器来解释，并用于实现一个可折叠面板。下面是该新元素的一个示例：

```
<details open="true">
   <summary>Drill down</summary>
   <div id="details_inside">
         This text was initially kept hidden from view
   </div>
 </details>
```

open 特性会指示你是否要对内容进行初始显示(参见图 12-1)。<summary>元素会为可单击的占位符指定文本，剩下的内容会被隐藏起来或按需显示。在撰写本书时，谷歌 Chrome 和苹果 Safari 浏览器就成为了少数几个支持此功能的浏览器。但 Internet Explorer 11 并不支持。

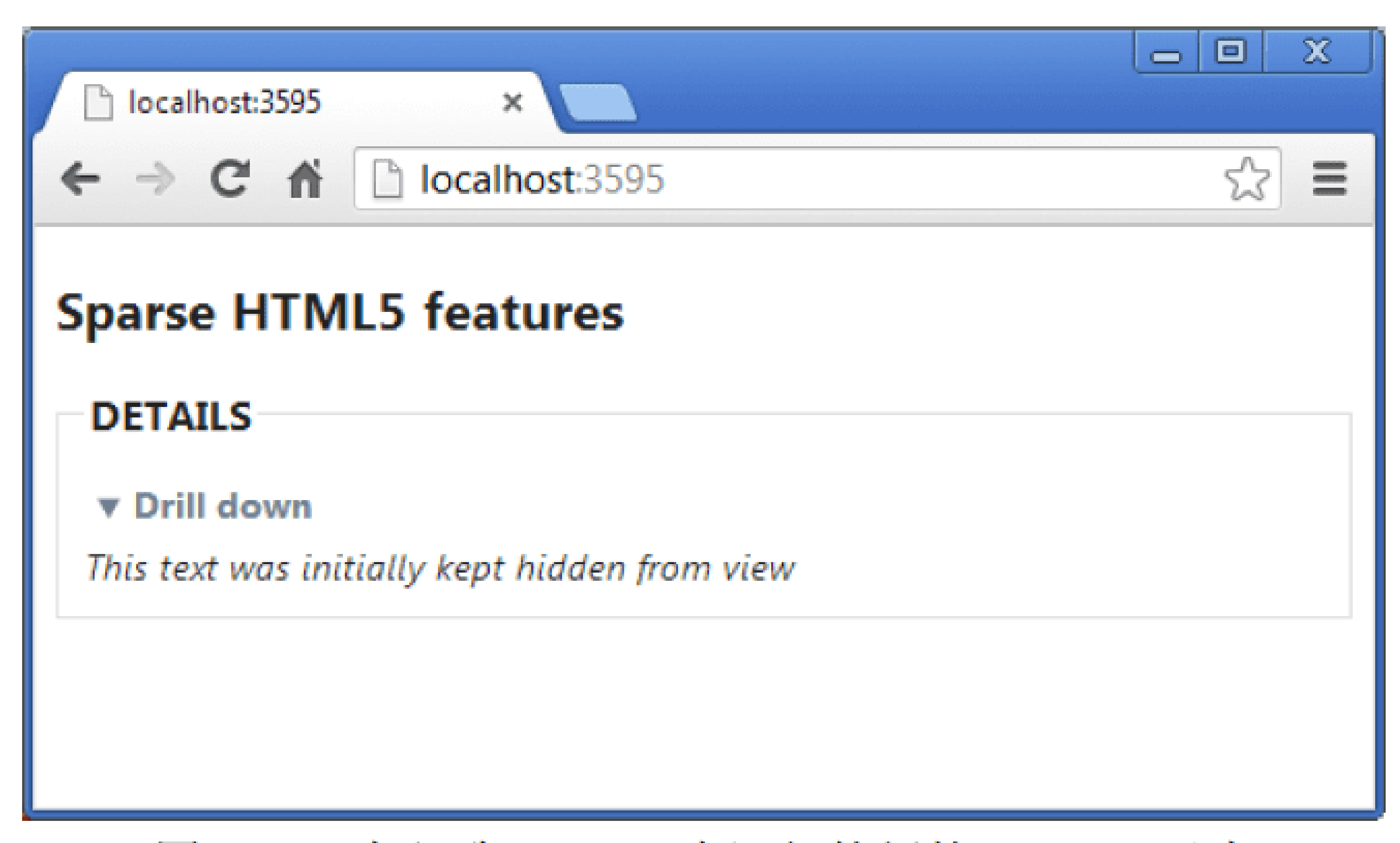


图 12-1　在谷歌 Chrome 中运行的新的<details>元素

注意：

如果想支持能够兼容 HTML5 的浏览器，同时保持与旧浏览器的兼容，那么你需要同时使用新的 HTML5 标签和替换标签；旧的浏览器会直接忽略掉新的 HTML5 标签。

你在图中看到的图标是由浏览器提供的。<details>元素需要一点 CSS 来增加美观性。下面是用于图 12-1 中所示元素的 CSS：

```
<style>
summary {
   padding: 5px;
   font-weight: bold;
   color: #708090;
}
#details_inside {
   font-style: italic;
}
</style>
```

注意：

HTML5 移除了一些只会增加冗余的没有多大用处的元素。这些不再受支持的元素中最引人注目的是<frame>(不过 IFRAME 元素仍存在)和<font>元素。此外，还移除了几个样式元素，比如<center>、<u>和<big>。其原因是这些功能可以通过 CSS 轻松地实现。出于某些原因，现行草案保留了类似<b>和<i>这样的元素。

4. 新的输入类型

目前，HTML(其次是浏览器)只支持纯文本作为输入。日期、数字，甚至电子邮件地址之间均有不少差异，更不用说预定义的值了。如今开发人员要负责通过实现输入文本的客户端验证来避免用户输入不必要的字符。jQuery 库有几个简化该任务的插件，但这正好强化了一个观点——输入是一个棘手的问题。

HTML5 带有大量的用于<input>元素特性类型的新值。另外<input>还依赖几个主要与这些新输入类型相关的新特性。下面是几个例子：

```
<input type="date" />
<input type="time" />
<input type="range" />
<input type="number" />
<input type="search" />
<input type="color" />
<input type="email" />
<input type="url" />
<input type="tel" />
```

这些新输入类型的实际效果是什么呢？虽然没有完全标准化，但预期的效果是浏览器提供特设的 UI，用户可以轻松地输入日期、时间或数字。其中的一些新输入类型专门针对手机

网站用户的需求。尤其是，email、url 和 tel 类型会推动智能手机上的移动浏览器(比如 iPhone、Android、Windows Phone)自动调整键盘的输入值范围。图 12-2 显示了在 iPhone 上的 tel 输入字段键入字符的效果：键盘默认为数字和与手机有关的符号。

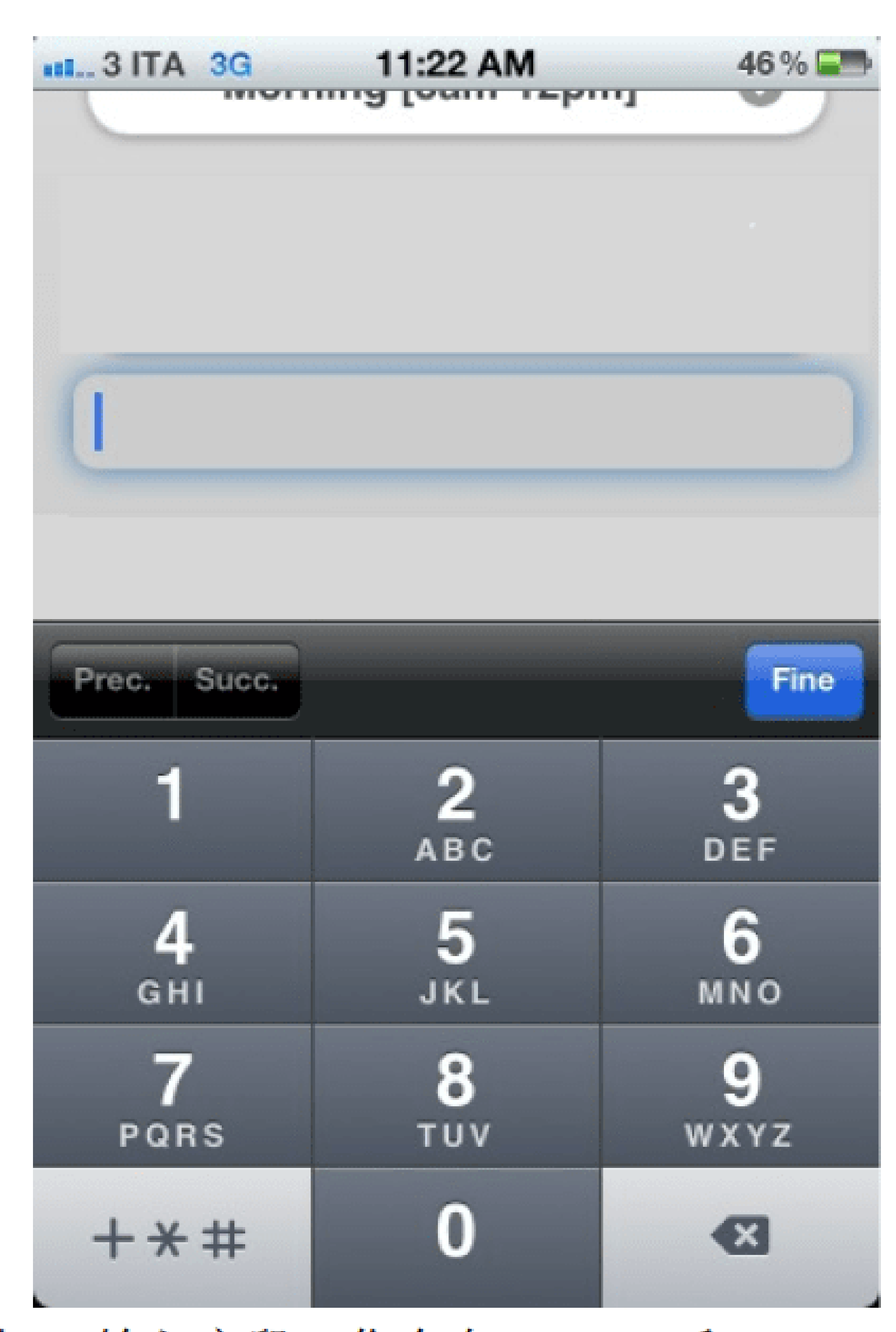


图 12-2　iPhone Safari 上的 tel 输入字段。你会在 Android 和 Windows Phone 上发现类似的实现

当前，不是所有浏览器都提供相同的体验，而且虽然它们基本上对与各种输入类型相关联的 UI 处理是一致的，但仍存在着一些主要差别，这可能需要开发人员添加基于脚本的自定义填充。举个例子，我们考虑一下日期类型。在撰写本书时，由 Opera 提供的界面与你在 Chrome 中看到的就不一样(见图 12-3)；Internet Explorer11 则没有提供任何对日期的特殊支持。

一般来说，最新智能手机上的移动浏览器都是兼容 HTML5 元素的。尽管如此，作为移动网站的开发人员，在使用新的像 email、number、url、date、tel 这样的输入元素时，可能仍然需要谨慎，最好对所有内容小心处理！

最后，值得注意的是，实现了大家期盼已久功能的占位符特性，会在可能的文本框中显示提示，而范围和日期/时间输入字段则不会显示提示文本。

注意：

归根结底，除非浏览器主动统一地支持新的输入字段，否则开发人员仍需要做一些有关 JavaScript 填充的工作，确保正确的数据提交到服务器，从而使用户正确了解哪里出了错。移动端页面比桌面页面更多地受益于 HTML5 兼容的浏览器。

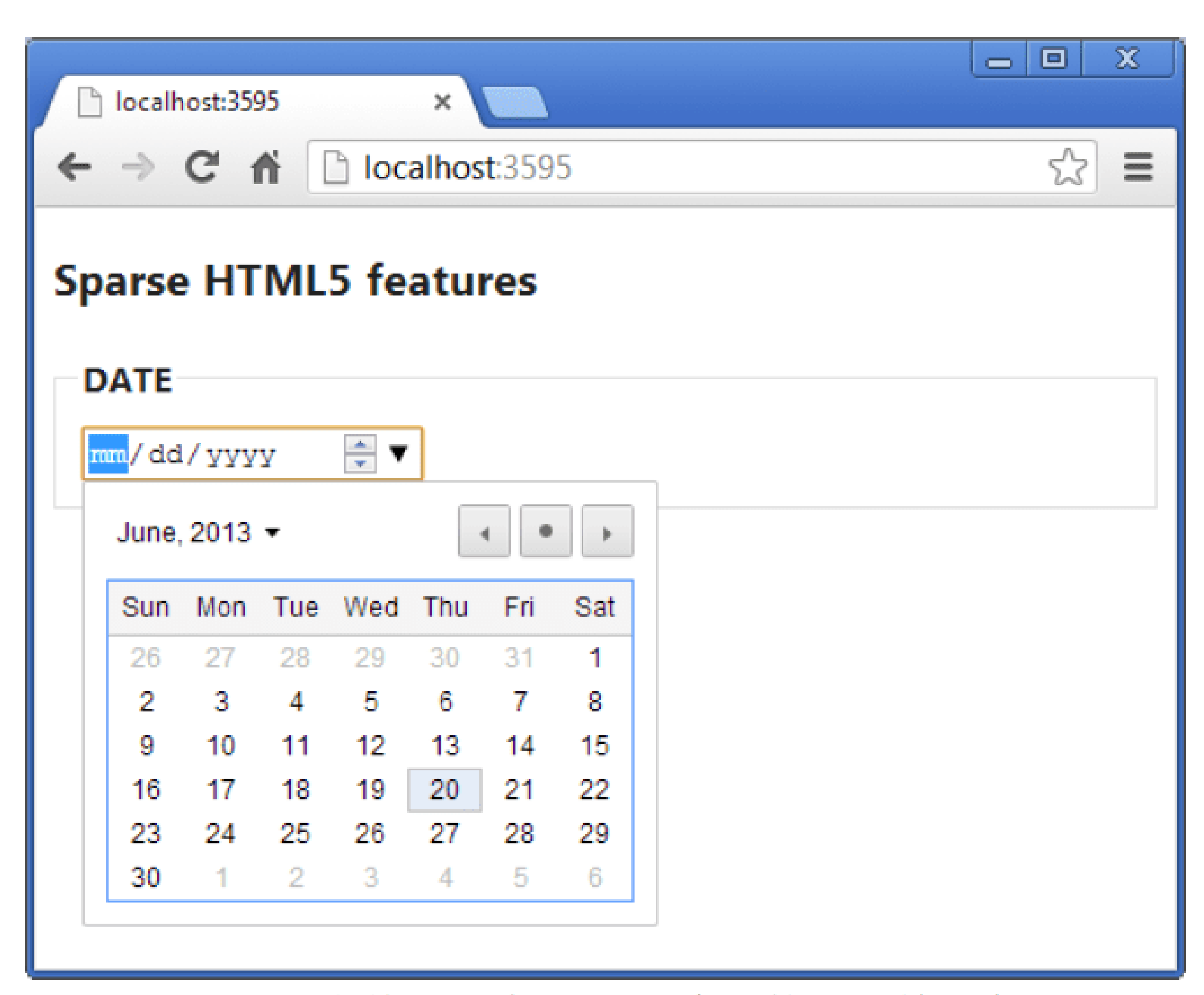


图 12-3　当前由谷歌 Chrome 实现的 date 输入字段

5. <datalist>元素

HTML5 表单中的另一个非常好的改进之处是<datalist>元素。该元素是流行的<select>元素的特殊版本。它提供了与<select>元素相同的行为，但下拉列表是应用于文本输入字段的。下面是一个示例：

```
<input list="countries" />
<datalist id="countries">
  <option value="Italy">
  <option value="Austria">
  <option value="Australia">
  <option value="Albania">
  <option value="Sweden">
  <option value="Denmark">
</datalist>
```

实际影响是，当输入字段接收到焦点时，菜单即会显示出来，用户可以输入任意文本，也可以选取一个预定义的选项。图 12-4 显示了谷歌 Chrome 浏览器提供的该功能。

6. 本地存储

HTML5 提供了一个标准的 API，使用户在设备上存储数据更可控，代表了对 cookie 的有效替代。本地存储的平均大小为 5 MB 左右，比 cookie 更大。

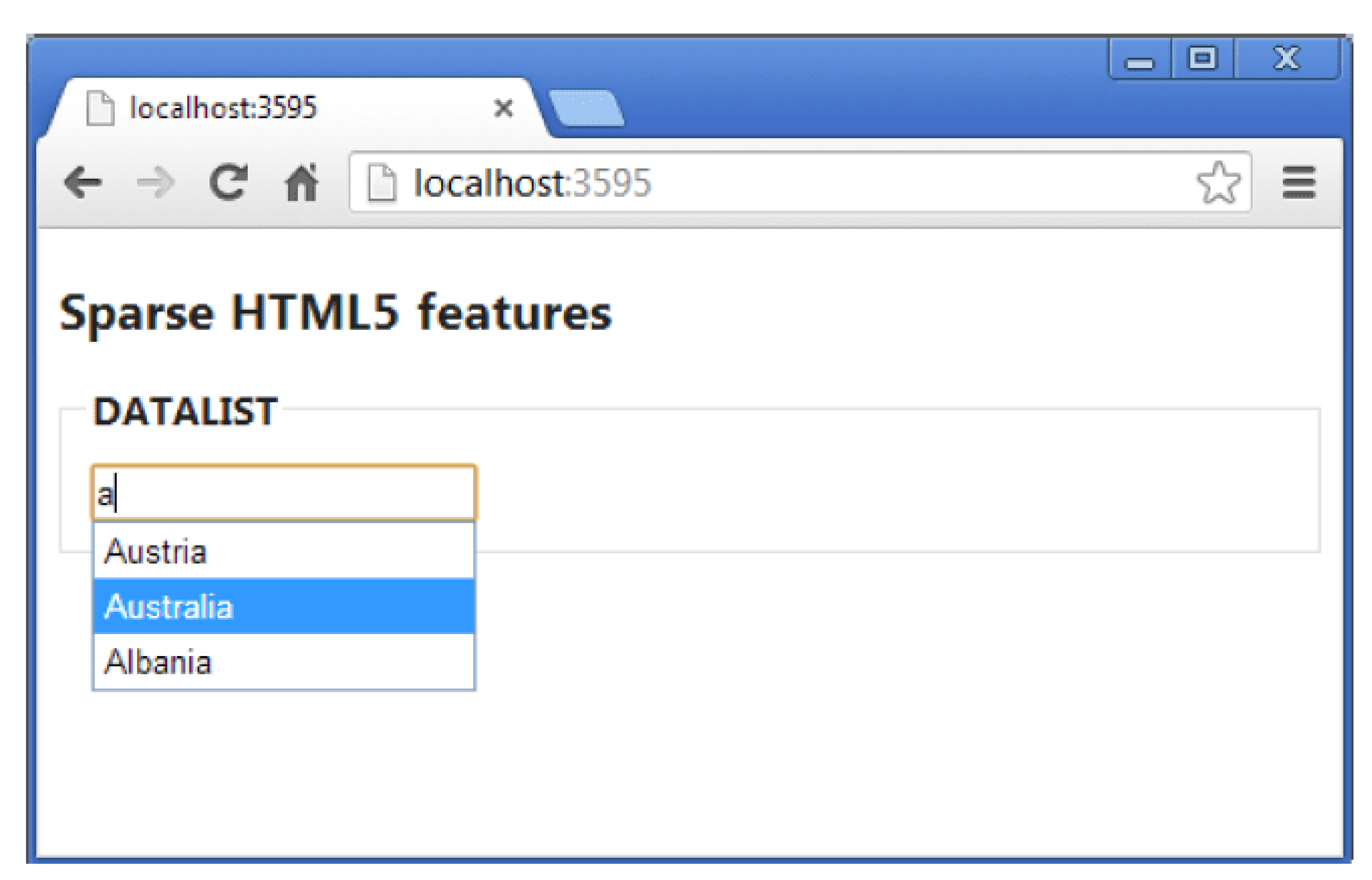


图 12-4　运行中的<datalist>元素

你要通过由浏览器窗口对象公开的 localStorage 属性来访问本地存储。localStorage 属性提供了一个基于字典的编程接口，类似于 cookie 的编程接口。可使用方法来添加和移除项、计算存储中的项数、获取特定项的值，以及清空存储。下面显示了如何保存一个值以及稍后对其进行检索：

```
<script type="text/javascript">
function save() {
  window.localStorage["message"] = "hello";
}
function init() {
  document.getElementById("message").innerHTML =
    window.localStorage["message"];
}
</script>
```

加载后，页面会检索并显示来自本地存储(如果有的话)的数据。这些数据通过一个以交互方式调用的函数被保存到存储器。保存到本地存储器的数据会无限期保留在用户的设备上，除非你以编程方式清空它。该存储是特定于应用程序的。

除了 localStorage，HTML5 还提供了一个 sessionStorage 对象，它具有相同的编程接口，但会保存到浏览器的内存。sessionStorage 对象会在当前浏览器会话结束时被清空。Web 存储接受基本类型(但规范在这一点上并没有限制，因此可以接受任何 JavaScript 类型)，但如果序列化为 JavaScript 对象标记(JSON)格式，也可以保存复杂的对象。

7. 音频和视频

HTML5 的最大好处之一是告别了(但真的能说再见吗？)仅为播放音频和视频的外部插

件，如 Flash 和 Silverlight。HTML5 带来了两个新元素<audio>和<video>，它们指向 URL 并播放任何内容。这些标签的浏览器实现也预期会为用户提供一个可以暂停和恢复播放的控制栏。下面显示了如何链接一个音频资源。

```
<audio poster="init.png" controls="controls">
   <source src="nicestory.wav" />
</audio>
```

多媒体元素(主要是指视频)的硬伤是文件的格式，既包括文件格式，也包括编码解码器。HTML5 标准不会对编码解码器进行正式调用，所以对有关格式的支持将依然取决于供应商。从开发人员的角度来看，这并不是好消息，因为它代表了一个分歧点；不同的浏览器支持不同的格式，而你需要检测浏览器或为浏览器提供多个可供选择的文件。下面是指示选择视频格式的语法：

```
<video poster="init.png" controls="controls">
  <source src="tiger.mp4" type="video/mp4" />
  <source src="tiger.webm" type="video/ogg" />
  Oops, it seems that your browser doesn't support video.
</video>
```

注意，你要使用 controls 特性来显示控制栏，并使用 poster 特性来指定用作启动画面的图片，直到媒体准备好播放。

流行的编码解码器是 MP4、MOV 和 AVI。你应该为 Internet Explorer 和 Safari 准备 MP4 编码的视频，而为其他的浏览器准备 OGG/Theora。目前看来，这似乎是避免外部插件的完美解决方案。但是，因为这是一个变化频繁的问题，所以最好还是三思而行。

12.1.2　RWD

大多数开发人员和技术管理人员还清楚地记得仅仅十年前构建用于各种浏览器的网站的噩梦。比如有一段时期，Internet Explorer 有一个功能集，它与 Firefox 或者 Safari，甚至是它自己的早期版本都不相同。那真的把页面标记的编辑工作弄得乱七八糟。据说目前大约 70%由 jQuery 组成的代码是用来收拾旧浏览器的残局的，最显著的就是 Safari 和 Internet Explorer 的怪异模式。甚至可以说，开发人员已经开始遗忘“浏览器大战”了，因为 jQuery 攻下了这块地盘；另一方面，浏览器大战无疑是促成 jQuery 快速普及的因素之一。

在移动领域，不同设备需求的数量级是以千为单位的，不似不久前的桌面浏览器的数量那么少。如果十年前浏览器的混乱程度让你感到惊惧，那么面对今天数量庞大得多的各种移动设备又该怎么办呢？这是真正的痛苦之源。铭记于此，开发人员学会了要把重点放在有效的功能上，而不是专注在与浏览器类型和名称相关的通用性能上。然而，这一原则理解起来简单，用于实践却很难。这正是 RWD 的出发点。

1. 特征检测

通过后面示例中的横向思维，RWD 才应运而生。设备检测很难？好吧，就不要那么做。你抓住客户端上可用基本信息的几个片段(比如浏览器窗口大小)，设置特设的样式表，让浏览器在页面中相应地重排内容。

你要停止检测请求设备的功能，并根据可以在设备上以编程方式检测的内容来决定显示的内容。这种方式广泛依赖大量的浏览器技术——主要是 CSS 媒体查询——且带来了明显的好处，作为开发人员，你只需要设计和维护一个网站即可。灵活适应内容的担子就落到了图形设计师或像 Twitter Bootstrap 这样的特设库身上。

功能检测的主要优势，我们可以总结为“一个站点适合所有情况”，却也可能成为最主要的弱点。你真的只想要一个网站吗？你真的想对智能手机、平板电脑、笔记本电脑和智能电视提供“相同”的网站吗？这个问题的答案总是要针对于具体的业务。总体而言，只能说“视情况而定”。

让我们首先了解 RWD 的本质，然后探讨它可能的缺陷。

2. CSS 媒体查询

让 RWD 焕发生命力的是 CSS 媒体查询。媒体查询在 CSS 3 中引入，是开发人员定义带条件的 CSS 样式表的语法，在窗口调整大小或发生其他一些系统事件时浏览器就会动态加载该 CSS 样式表。CSS 媒体查询使开发人员能够轻松地创建多视图页面，它可以通过不同尺寸屏幕的设备来使用，从 24 英寸的桌面显示器到大部分 3 英寸屏幕的智能手机都行。

注意：

需要指出的重要一点是，CSS 媒体查询不是专门针对移动开发的一种技术。然而，其内在能力和解决方案的灵活性使得它也很适合用在手机网站的建设上。

对开发人员来说，CSS 媒体查询的主要优势是明显的。开发人员只需要编写一组页面和一个后端。这组页面针对的是你打算支持的最宽可能屏幕——通常是桌面大小——并存储尽可能多的元素。接着，设计师要提供多个 CSS 文件，每一个支持一个中等大小屏幕的尺寸。中等大小屏幕的尺寸称为布局断点。最常见的情况是，你有一个断点是 480 像素，另一个是 800 像素，可能还有更多个断点。当用户将浏览器窗口的宽度调成低于 480 像素时，浏览器就会自动选择用于 480 像素的 CSS。当窗口宽度调为介于 480 像素和 800 像素之间时浏览器也会自动选择；在这种情况下会选择用于 800 像素的 CSS。当相同的页面是在智能手机中浏览时，会自动应用用于小屏幕的 CSS(480 像素)；而平板电脑可能会得到 800 像素的布局。这种方式简单而有效。

通过添加另一个断点和相关的 CSS 文件，可以创建一个屏幕大小介于 480 像素和 800 像

素之间的特设视图。这样，你还可以解决迷你平板的问题。你需要支持像智能电视这样的大屏幕吗？没问题。只需要添加另一个 CSS 文件即可。真的很容易，它非常好用，虽然设置一个能用于现实中复杂页面的 RWD 解决方案没那么简单。

3. 运行中的 CSS 媒体查询

你想支持多少种不同的屏幕分辨率？答案取决于预期的受众和要呈现的内容。不过，宽度调整为 400 像素的桌面浏览器和屏幕大小相同的智能手机之间却有着巨大的差异。就计算能力和资源方面而言，笔记本电脑是一回事；智能手机完全是另一回事。不过，CSS 媒体查询无法基于单台设备进行区分。

我们假设现在这不构成问题，再进一步假设从业务的角度出发，专注于视窗的大小是可以接受的。首先，你需要决定打算为多少个分辨率使用不同的布局。一个分类示例可能由以下断点构成：

- 最大 480 像素
- 最大 800 像素
- 大于 800 像素

对于每一个断点，你要创建一个不同的 CSS 文件来安排样式元素，包括将它们顺次放到屏幕底部或隐藏其中一些。这样一来，如果页面链接的是静态图片的话，你还可以决定引用较小的图片。

通过使用用于<link>元素的经典语法的轻微变体，你就能引用这些 CSS 文件：

```
<link type="text/css"
      rel="stylesheet"
      href="view480.css"
      media="only screen and (max-width: 480px)">
```

在这个例子中，view480.css 文件只有当页面呈现在屏幕上且浏览器窗口不大于 480 像素时才会用到。当使用媒体查询且未找到匹配项时，显然没有样式表会被应用到页面。

注意：

过去，media 特性表明 CSS 为何种媒体而设计——屏幕、打印机、电视、视频终端等。在支持全部 CSS 3 标准的现代浏览器中，media 特性的值可以包括一个选择媒体和一些运行时条件的查询。在 http://www.w3.org/TR/css3-mediaqueries 处可以找到关于媒体查询的全部文档。

CSS 媒体查询语言是基于一对布尔运算符——and 和 not——以及一些浏览器属性的。表 12-1 列出了你能用来选择最适合样式表的浏览器属性。

表 12-1　构建 CSS 媒体查询的属性

浏览器属性	描　　述
设备宽度、设备高度	物理设备屏幕的宽度和高度
宽度、高度	呈现视窗的宽度和高度；例如，浏览器窗口
方向	当高度大于或等于宽度时返回纵向。否则返回横向
高宽比	表明宽度与高度之间的比例；例如“16/9”
设备高宽比	表明设备宽度与设备高度之间的比例；例如“16/9”

请记住，设备宽度和设备高度、以及宽度和高度属性也支持 min/max 前缀。

通常在媒体查询表达式的开头你会发现关键字 only，但它实际不具有功能性作用。添加这个关键字的唯一目的是从媒体查询语句中排除旧的浏览器；当媒体类型带有 only 前缀时，旧的浏览器就不能理解该媒体类型，并且将直接忽略该语句。

如前面所示，可以在宿主页面的<link>元素内部使用媒体查询表达式。这样一来，你最终会让每个断点使用一个单独的 CSS。你还可以创建包含多个媒体节的单个 CSS 文件，如以下代码所示：

```
@media screen and (min-width: 480px) {
    body {
        background: yellow;
    }
    ...
}
@media screen and (min-width: 800px) {
    body {
        background: blue;
    }
    ...
}
```

注意：

无论是创建单个文件还是多个文件，都应考虑到在 RWD 解决方案中，你要处理大量的 CSS 设置，大部分都是重复性的。为此，查看一下像 LESS(http://lesscss.org)这样的动态样式表框架是值得的。使用这些类型的框架，你就能使用像变量和函数这样的友好编程结构来动态创建 CSS 设置。

在表 12-1 中，有两个看起来类似的属性：宽度和设备宽度。如前所述，前者指的是浏览器的宽度，后者表示的是设备的屏幕宽度。为了自适应呈现，你应该一直使用宽度。不过，在移动设备(智能手机和平板电脑)上，任何应用程序都可以在全屏模式下使用，所以这两者

其实并没有实际区别。在这一点上，Windows 8 平板电脑是一个值得注意的例外设备，因为除了全屏模式，应用程序还可以在分屏和填充模式中运行。

注意：

在 CSS 媒体查询等级 4(http://dev.w3.org/csswg/mediaqueries4，这是下一个标准版本)中，会添加一些有助于区分移动设备和桌面设备的新属性。不过，这似乎并不是设备性能集，开发人员不需要真的为其制定面向移动端视图。

4. 流动布局

单独使用 CSS 媒体查询并不能真的实现响应式布局，但它确实在某种程度上有助于响应动态修改的条件。如果设计只是为了迎合一些预设断点的话，那么用户将得到相同的布局，无论其浏览器的宽度是否介于两个断点之间。例如，假设你使用了以下设计：

```
@media screen and (min-width: 480px) {
    body {
        background: yellow;
    }
    #container {
        width: 480px;
    }
    ...
}
@media screen and (min-width: 800px) {
    body {
        background: blue;
    }
    #container {
        width: 800px;
    }
    ...
}
```

当浏览器宽度达到 480 像素时你有一个断点——此时，背景将变成黄色且容器元素将设置成 480 像素。可以扩大浏览器窗口，但只有在其宽度达到 800 像素时你才能注意到变化。图 12-5 描绘了 600 像素时的一个屏幕截图。

由于该布局使用了固定度量值，因此我们最终会发现一些空的未使用的空间。可通过添加更多的断点并编辑和保持更多 CSS 文件来消除未使用空间的影响。然而，对于非测试站点，你实际上不能处理 3 个或 4 个以上的断点。

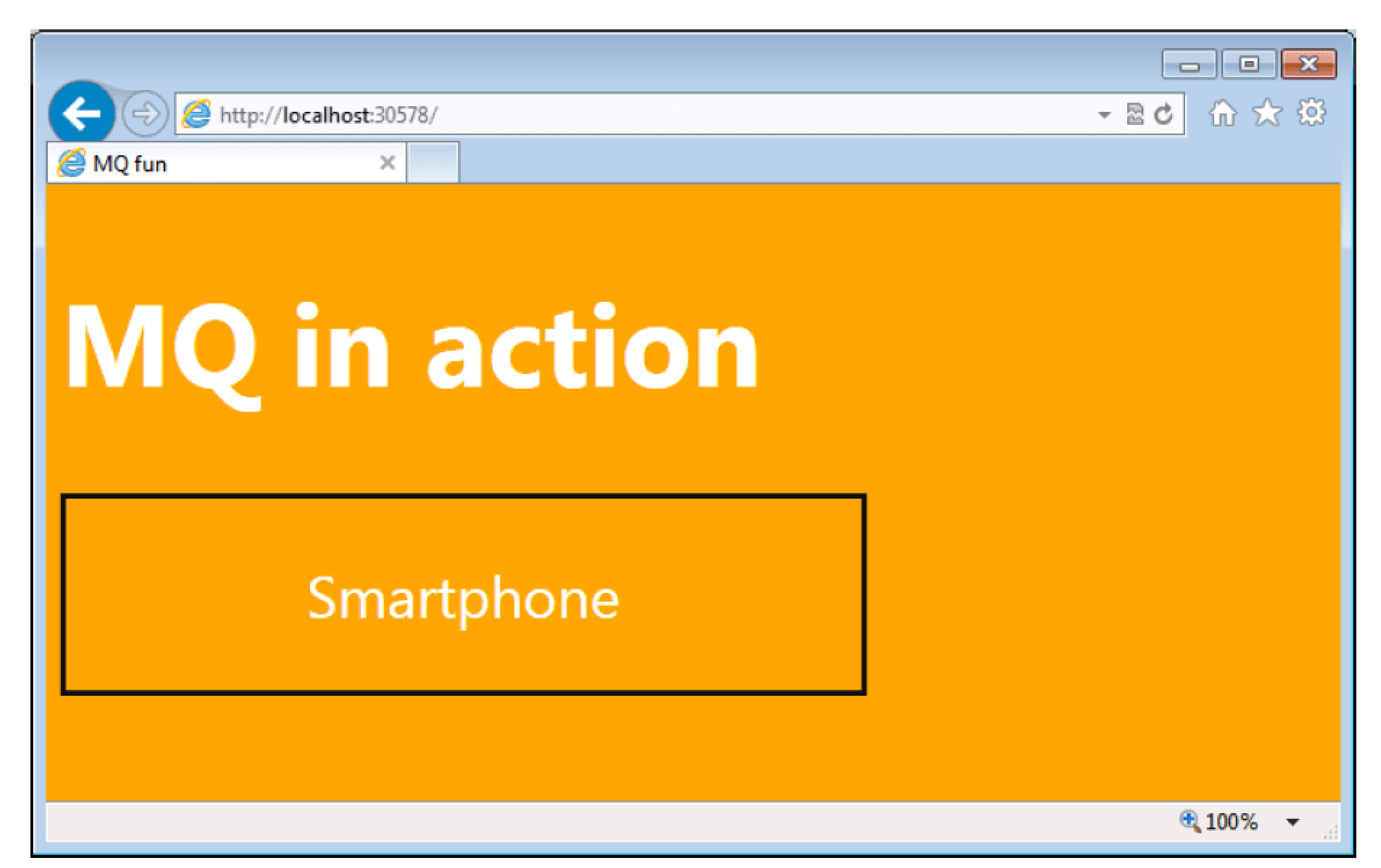


图 12-5　一个自适应但并非完全响应式的视图

一个真正的响应式布局是一个可以适应浏览器窗口的宽度和/或高度任意变化的布局。要达成此效果，你需要使用基于字符长度或比例的 CSS 度量值来构建你的布局。以这种方式，你的设计可以几乎不受限地放大和缩小。这有时也称为流动布局或按比例布局。

在 CSS 中，可以使用一些单位来设置宽度、高度和字体大小。有一个单位正变得越来越流行，这就是字符长度。一个字符长度就等于当前的字体大小；接着，“1.2 字符长度”会将当前字体大小增加 20%。像素和点都是固定单位，且不能随窗口大小而缩放。然而，比例和字符长度都是相对度量值——尽管其相对的内容不同。字符长度单位总是相对于字体大小的，而比例是相对于包含块的(例如，<body>或<div>)。也可以将比例应用到字体大小。在这种情况下，它表示的是与父字体大小相关的变化。通常，我认为使用比例来表示 Web 元素的尺寸(块和文本)更加可靠，并且可以得到跨浏览器的一致性。

话虽如此，但流动布局主要是为了通过相对度量来表示位于布局中的内容。

5. 当 RWD 遭遇移动端

RWD 绝对是一种 Web 设计的强有力的方法。它会以两种方式向你提供帮助：首先，它使你的站点看起来更棒，而不用在乎浏览器的大小；其次，通过向 Twitter Bootstrap 这样的框架，它还能让非设计者得以快速创建很好的模板。不过，RWD 却并非为专门服务移动设备而设计的，但它的功能强大而灵活，所以也能用于几乎所有移动设备上的自适应页面视图。

移动设备的一个关键特征是其屏幕更小——智能手机大约 400 像素，而平板电脑大约

800~1000 像素。当 RWD 将你的视图很好地呈现在那些尺寸的屏幕上时，你的设置工作就算完成了——对吗？

RWD 需要 CSS 和 CSS 媒体查询才能工作。使用 CSS，可以完成很多任务，但 CSS 与编程并无关系。CSS 媒体查询属性会告知你关于设备的一些信息，但它不能提供你需要知道的所有信息。例如，有关操作系统的信息，以及设备是否是移动手机、平板电脑、智能电视、或是网络机器人等，这些你仍然一无所知。

例如，RWD 会告知你当前托管在视窗上的页面宽度是 800 像素。但是，它不能向你表明该视窗属于平板电脑还是属于可缩放大小的 Internet Explorer 桌面窗口。有时候，这是给最终用户带来相当大差距的细节；随之对于开发人员也是如此。因此，问题就变成 RWD 到底在哪些方面能够符合移动端方案？当你要创建一个移动端网站时使用 RWD 又是否真的值得信赖呢？

移动设备与典型的个人计算机不同。它的屏幕更小——有时是很小的屏幕。它并不具有相同的计算能力和存储；它并不具有相同的电源；且它通常都是可触摸的(这对于桌面端来说通常无法做到，尽管较新的型号可能会使用触摸屏显示器)。再者，移动设备通常是在移动中使用的，这是为了快速迅即地处理事务。因此，其连接可能会发生在任何时候，并且有时连接速度可能会较慢和不稳定。

当用户正在使用移动设备时，他们会想通过一两次单击就找到选项和操作；他们很可能不需要大量的功能和信息。有时候他们需要的信息或信息聚合与适合所有站点的那些信息是不同的。移动端用户可能需要不同的运行指引并且肯定需要你仔细制定应用程序的用例。

RWD 是采用为桌面端设计的站点并让该站点在各种移动设备上良好呈现的一种极佳方式。但即使这样，你最终也未必能得到一个为移动设备而优化的站点。许多开发人员有时会宣称以移动优先的方式来进行设计，但他们所做的实际上是为桌面端进行设计，然后仅仅将该设计适应到移动端而已。我认为这肯定不是移动优先的设计方式！

注意：

RWD 能够很好地用于像门户这样的网站。而对于具有高度可交互性、实现工作流(比如一个预订站点)且充斥着用户必须填充的表单的站点就不能达到相同的适用性了。

12.1.3　jQuery Mobile 的执行摘要

jQuery Mobile(jQM)是在流行的 jQuery 库上工作的，它是全新构建出来的，旨在为构建移动站点提供一个全面的平台。通过这个库你能够以你的方式进行编码，而且这个库会负责在浏览器上用最可行的方式呈现标记。通过使用 jQM，你完全不必担心设备检测和性能方面的问题。该库会确保输出结果在低级浏览器上同样有效；无论得到的输出结果是否真是你想要的……而这就是另外一个问题了。

ASP.NET MVC 开发人员可以在微软 Visual Studio 的移动项目模板中找到特有的 jQM 库。这一事实似乎在推广一个愿景，即 jQM 是在网站中启用移动端支持的一种必要条件。和往常一样，这也是要看情况的。

简单来说，jQM 就是一个用于呈现的 JavaScript 库，它清楚如何将普通的 HTML 标记转换成移动视图。这意味着页面中的按钮和输入字段会以某种方式进行呈现，该呈现方式会模拟类似于 iOS 这样的流行移动平台的可视化元素的外观和感觉(以及部分行为)。jQM 的简单易行给你的页面增添了移动端的外观与感觉。但这并不意味着你的站点会突然变成精心设计的移动站点。

重要提示：

可以愉快地使用 jQM(或者像 Kendo UI 或 Sencha 这样的其他供应商指定的框架)。不过，如果将站点设计成能从移动设备轻松舒适地使用的话，就能从根本上让你的客户感到满意。这也是重新制作用例和数据聚合的问题。在这方面没有哪个库能帮上你。

jQM 的官方站点是 http://www.jquerymobile.com。你还可以从 NuGet 或直接在 http://code.jquery.com 处得到最新的版本。

1. 主题和样式

如果没有一个关联 CSS 文件的话，jQM 库几乎是不可用的。该库承担了页面上大量的工作，并将页面从普通的<div>标签集合转换成对移动端友好的悦目和可用的文档。为了实现这一目标，必须遵循严格的标准创建大量的样式和图片。该库有一个预定义的 CSS 文件要包括进来。像许多其他内容一样，主题当然也是由开发人员来自定义的。

jQM 库带有一些预定义的主题，可通过字母表的首字母进行识别：a、b、c 等。每个主题都由被统一应用到各种 HTML 元素的若干 CSS 样式构成。大多数时候，只需要选取你想要的主题；而选择是基于颜色进行的。以下代码显示了如何设置主题。

```
<div data-role="page" data-theme="b">
...
</div>
```

通过使用 data-theme 特性，你能够将不同主题应用到页面的不同部分(稍后将对 data-*特性进行更多介绍)。主题应用是默认的；不过，通过在特定元素上使用普通 CSS 命令，你就可以重写其设置，如下所示：

```
<div data-role="footer" class="my-footer center">
...
</div>
```

上面的<div>元素被赋予了页脚的角色。就其本身而言，它会从库中获得一个特定样式，这取决于当前的主题。不过，class 特性是用来重写一些属性的(颜色、边框、字体等)。class 特性很难完全替换设置；如果你的目的是完全替换，可能最好的方式还是使用 jQM 的 ThemeRoller 工具来创建你自己的自定义主题。

2. Data-*特性

在 HTML5 中，data-xxx 特性是自定义特性，可以用来更好地定义元素的语义。这样的特性会被强制使用 data-xxx 形式的名称，但是框架(或页面)要负责指定该 xxx 可变部分，更重要的是，要负责其解释。data-xxx 特性总是返回和接受字符串。

jQM 库能识别不少 data-xxx 特性并使用它们来修饰 HTML 元素及赋予其特殊含义。由这些特性表示的语义能确定图表结果。

jQM 中最重要的一个 data-*特性是 data-role 特性。它表明了页面上下文中特定元素所发挥的作用。该特性通常用于修饰<div>标签，并将其作为特定的语义组件传递，比如页眉、页脚或内容。

3. jQM 中的页面

在 jQM 中，页面可以是单个 HTML 文件，也可以是已有页面的内部部分。在该库的术语中，这两种形式被称为单页面模板和多页面模板。非常棒的是，该库使你可以导航到页面，而不用在意其本质，无论它是物理页面(单个 HTML 文件)还是现有 HTML 文件的逻辑部分。以下是典型的页面定义：

```
<div id="homePage" data-role="page" data-theme="a" class="my-bkgnd">
...
</div>
```

页面是通过被设置成该页面值的 data-role 特性所修饰的<div>元素来定义的。它可以具有自己的 ID 并能够使用 class 特性来重写样式设置。通过使用 data-theme 特性，你就能选择用于整个页面的主题。jQM 页面通常包含页眉、页脚和内容。不过，应该注意的是，这只是一种惯例而已；页面可以包含任意有效的标记。一个 HTML 文件可以包含多个被标记为逻辑页面的元素，比如下面所示的这些：

```
<div id="homePage" data-role="page" data-theme="a">
...
</div>
<div id="aboutPage" data-role="page" data-theme="b">
...
</div>
```

当使用多个逻辑页面时，你应该会为每个逻辑页面使用唯一的 ID 以启用导航和初始化。

jQuery 开发人员都非常熟悉 ready 事件。该事件在页面 DOM 完全初始化之后就会触发，而页面编写者能够安全地完成页面元素的初始化。在 jQM 中，你不会使用 ready 事件；相反，你要使用的是新的 pageinit 事件。

```
<div id="homePage" data-role="page">
   <script type="text/javascript">
      $("#homePage").bind("pageinit", function () { alert("home");  });
   </script>
   ...
</div>
```

不同之处在于，在 jQM 中，Ajax 调用被用于隐式下载请求页面并设置动画效果和页面转换。这意味着页面(实际页面或虚拟页面容器)的显示遵循了与经典 jQuery 中不同的规则。

总而言之，你需要做的就是在页面元素内添加<script>标签并绑定表示具有 pageinit 事件的页面的<div>。此代码确保会在页面每次加载时被调用，无论它是由一个链接还是一个 Ajax 调用请求的。在 pageinit 中，你通常要注册你的 jQuery 插件并进行初始化工作(应用本地化字符串、设置控件等)。还有类似的事件可用于页面显示和卸载。

4. 页眉和页脚

data-role 特性常用的两个值是页眉和页脚。页眉栏包含页面标题和位于左侧及右侧的一对可选按钮(模拟 iPhone 模板)。页眉角色会接收由框架指定的样式并经受一些默认处理。作为开发人员，可以完全自定义页眉模板和文本以及按钮的目标。以下是一个示例页眉栏：

```
<div data-role="header">
  <h1>Home page</h1>
</div>
```

具体来说，第一个标题元素(h1 到 h6)用来给标题栏添加标题；如果其内容非空，则该文本也将变成页面标题，也就是会重写为<title>元素指定的任何值。只要标题元素是 Hx 元素，则用哪个标题元素给页眉栏添加标题都没关系——应用的样式也是如此。如果想单独为页面添加其自己的标题而不是页眉文本的话，则可以在页面容器上使用 data-title 特性。在页眉栏中出现的第一个链接会自动变成按钮样式并移动到左侧。

```
<div data-role="header">
  <h1>ASP.NET MVC</h1>
  <a href="..." data-icon="gear">Login</a>
</div>
```

但是，第二个链接会放置在右侧。如果只想在右侧放置一个按钮，可以添加一个额外的

class 特性，如下所示：

```
<div data-role="header">
  <h1>ASP.NET MVC</h1>
  <a href="..." data-icon="back">Back</a>
  <a href="..." data-icon="gear" class="ui-btn-right">Login</a>
</div>
```

data-icon 特性会在 jQM 主题中选择一个预定义的图标；ui-btn-right 值会将按钮移到右侧(在本示例中，这不是严格必须的，因为移到右侧的按钮是第二个按钮)。需要说明的是，如果使用 ASP.NET MVC 和 HTML 帮助器，则前面的锚点必须像下面这样重写(方法和控制器的名称可能会变更)：

```
@Html.ActionLink("Back", "index", "home", null, new {data_icon = "back"})
@Html.ActionLink("Save", "login", "account", null, new {data_icon="gear",
    @class="ui-btn-right"})
```

ASP.NET MVC 会自动将下划线(_)展开为破折号(-)，而@符号会转义 class 一词，否则这个词对于 Razor 编译器就是另一个意思了。

不同于页眉，页脚的目的并非是为了简化指定标记模板。不过，你在页脚区域放置的任意链接都将自动变成按钮，并且页脚中可以接受任何标记，包括表单标记。这样，你就能在页脚放置一个应用程序栏甚至是一个下拉列表，以便让用户进行选择，比如选择语言。注意 jQM 管理者页脚栏的实际位置。通过使用 data-position 特性，你就能保持页面底部的位置不变，如下所示：

```
<div data-role="footer" data-position="fixed" data-id="about">
...
</div>
```

当用户单击链接在页面间切换时，jQM 会将过渡形式应用到全部页面。有时，你会得到页脚保持不变的平滑效果。要实现这一点，页脚必须在进行过渡的两个页面中保持固定，此外，两个页脚必须具有设置为相同(唯一)标识符的 data-id 特性。图 12-6 显示了带有页眉、页脚和主体占位符的一个示例页面。

要完全自定义页眉模板，你需要将一个子<div>添加到页眉块并对其随意填充。请记住，如此一来你便失去了自动处理功能；例如，链接会显示为普通链接。你需要 data-role=button 特性来进行转换。

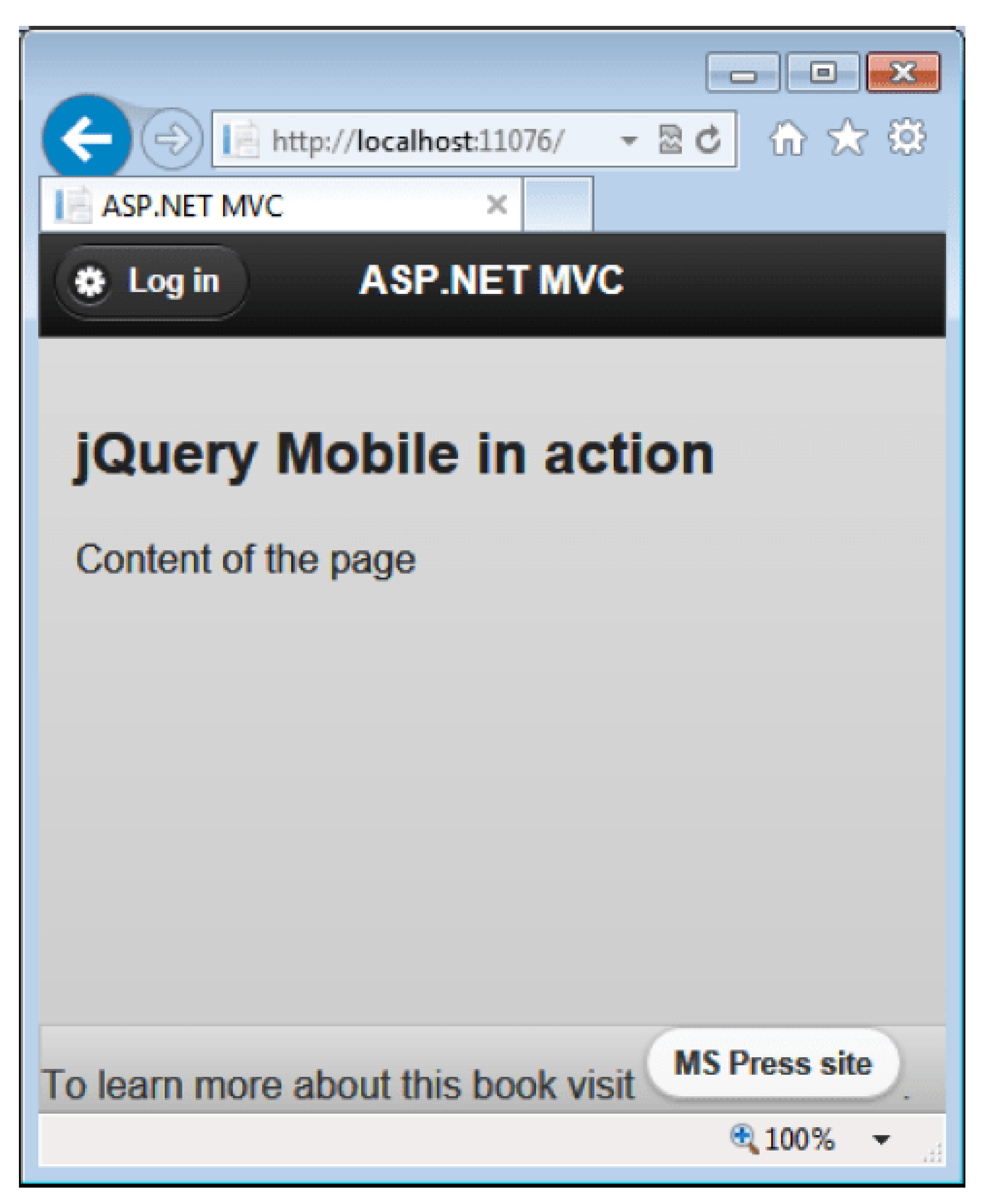


图 12-6　jQM 模板中的页眉和页脚

```
<div data-role="header">
   <div>
     <img src="@Url.Content("~/content/images/mobile.png")" alt="">
     <h3>Custom templates</h3>
     @Html.ActionLink("Back", "index", "home", null,
       new {data_icon = "back", data_role="button", @class="ui-btn-right"})
    </div>
</div>
```

5. 一切围绕列表

大多数移动站点的首页都应该仅提供操作的列表，就像大约三十年前的应用程序经典主菜单那样。你能够以各种方式呈现菜单项；可以用它们组成导航栏、按钮栏、磁贴集合，以及美观的普通链接列表。jQM 支持几乎任何方案，尤其用于导航栏和列表视图的效果极佳。

listview 角色提供了近来移动设备上最常见的 UI——非常接近于 iPhone 和 Android 的选择列表。以下是经由 jQM 处理过的普通 HTML 代码，会产生图 12-7 的结果。

```
<ul data-role="listview" data-inset="true" data-theme="c" data-
               dividertheme="a">
   <li data-role="list-divider">Chapter 12</li>
   <li>@Html.ActionLink("Read content", "Content", "Home")</li>
   <li>@Html.ActionLink("Run samples", "Samples", "Home")</li>
```

```
    <li data-role="list-divider">General</li>
    <li>@Html.ActionLink("About the author", "About", "Home")</li>
    <li>@Html.ActionLink("About the publisher", "About", "Home")</li>
    <li>@Html.ActionLink("Related books", "About", "Home")</li>
</ul>
```

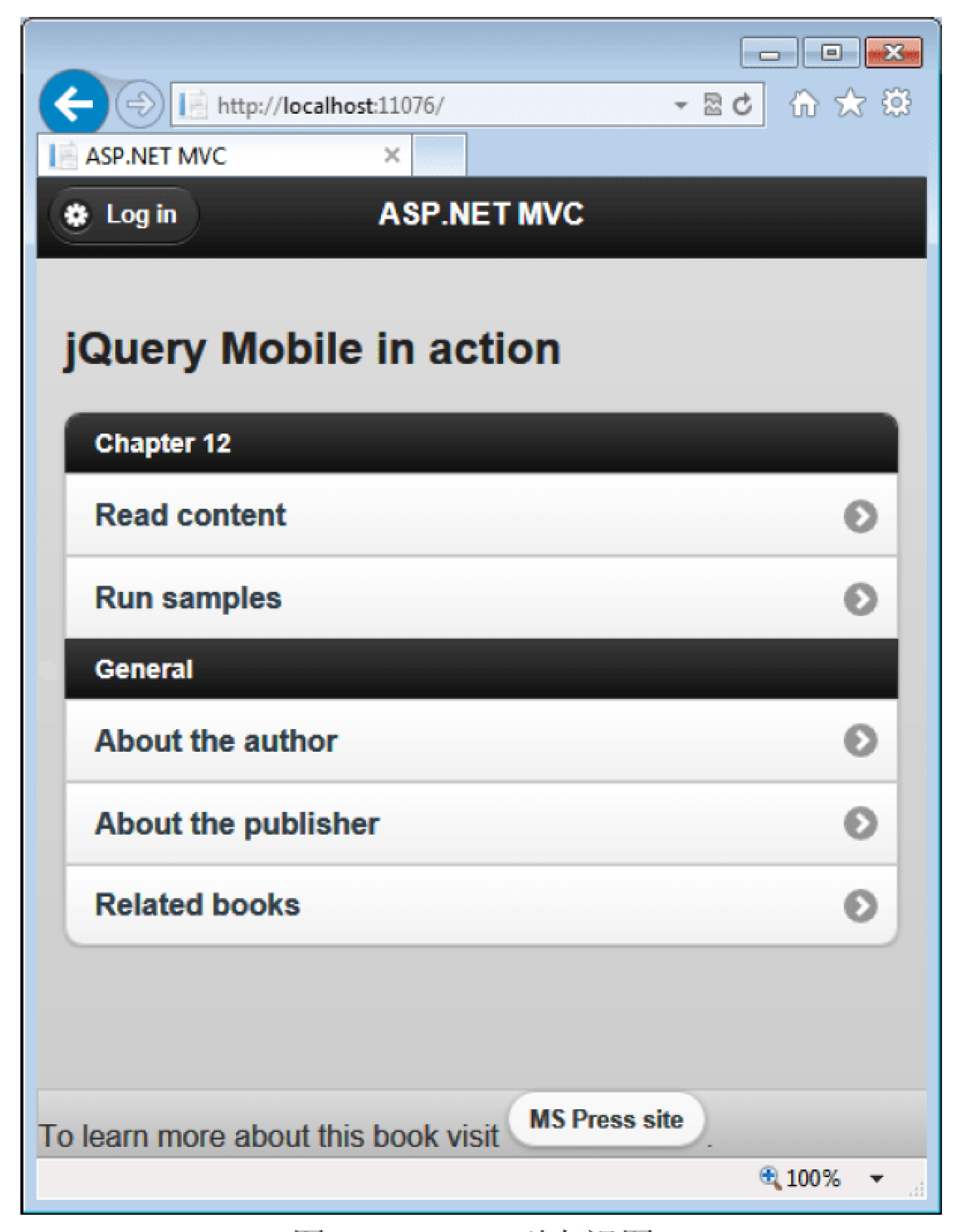


图 12-7　jQM 列表视图

通过使用<ul>和<ol>元素的各种不同变体，你就能创建编号列表和嵌套列表。图表化的变体包括 data-inset 特性和 data-filter 特性，data-inset 负责处理图 12-7 中所示的圆角和美观框架，而 data-filter 会添加一个搜索栏，该搜索栏能够自动补全静态添加到页面的列表项(该自动补全功能不需要连接远程服务来下载动态数据)。

可以将图标和图片添加到列表项。以下代码显示了如何将一个小图标添加到列表项的左侧：

```
<li><a href="...">
    <img alt="" class="ui-li-icon" src="..." />
    <span>Run samples</span>
```

```
</a></li>
```

ui-li-icon 类会确保图片左对齐；注意图片元素必须是锚点的子节点。同样，你还可以将文本添加到列表项的右侧。如果想要该文本在一个气泡内自动呈现，如图 12-8 所示，可以用 ui-li-count 类对该文本进行标记；或者，也可以使用 ui-li-aside 类。

```
<li>
    @Html.ActionLink("Related books", "Related", "Home")
    <span class="ui-li-count">3</span>
</li>
```



图 12-8　在 jQM 列表视图中显示气泡文本

6. 流动布局

尽管在布局中使用多个列对于移动站点来说很难称之为一个好主意，但若有一种简单方式将少量元素并排放置的确是非常棒的。在 jQM 中，你会找到非常基础但十分有效的 CSS 网格实现，以及能自动适应屏幕的流动布局。

该库提供了两个主要的 CSS 类：ui-grid 和 ui-block。前者将<div>(或<fieldset>)标记为容器网格，而后者将<div>标记为子元素。不过，主要的 CSS 类需要明显的文字来分别表明单元格和位置的数量。例如，ui-grid-a 类会将一行平均分成两块。第一块被分配给 ui-block-a 引用的内容；第二块被分配给 ui-block-b。下面是一个例子：

```
<fieldset class="ui-grid-a">
   <div class="ui-block-a"> First block </div>
   <div class="ui-block-b"> Second block </div>
   <div class="ui-block-c"> Third block </div>
</fieldset>
```

在这个例子中，还有分配给网格的第三块，该网格不能承载超过两块(它会将行宽分成 50/50)。实际效果是，样式为 ui-block-c 的第三个<div>会另起一行显示。要保持第三个子块位于同一行上，你要将网格样式修改成 ui-grid-b，它会将网格平均分成三部分(参见图 12-9)。依此类推，ui-grid-c 会分成四部分，而 ui-grid-d 分成五部分。

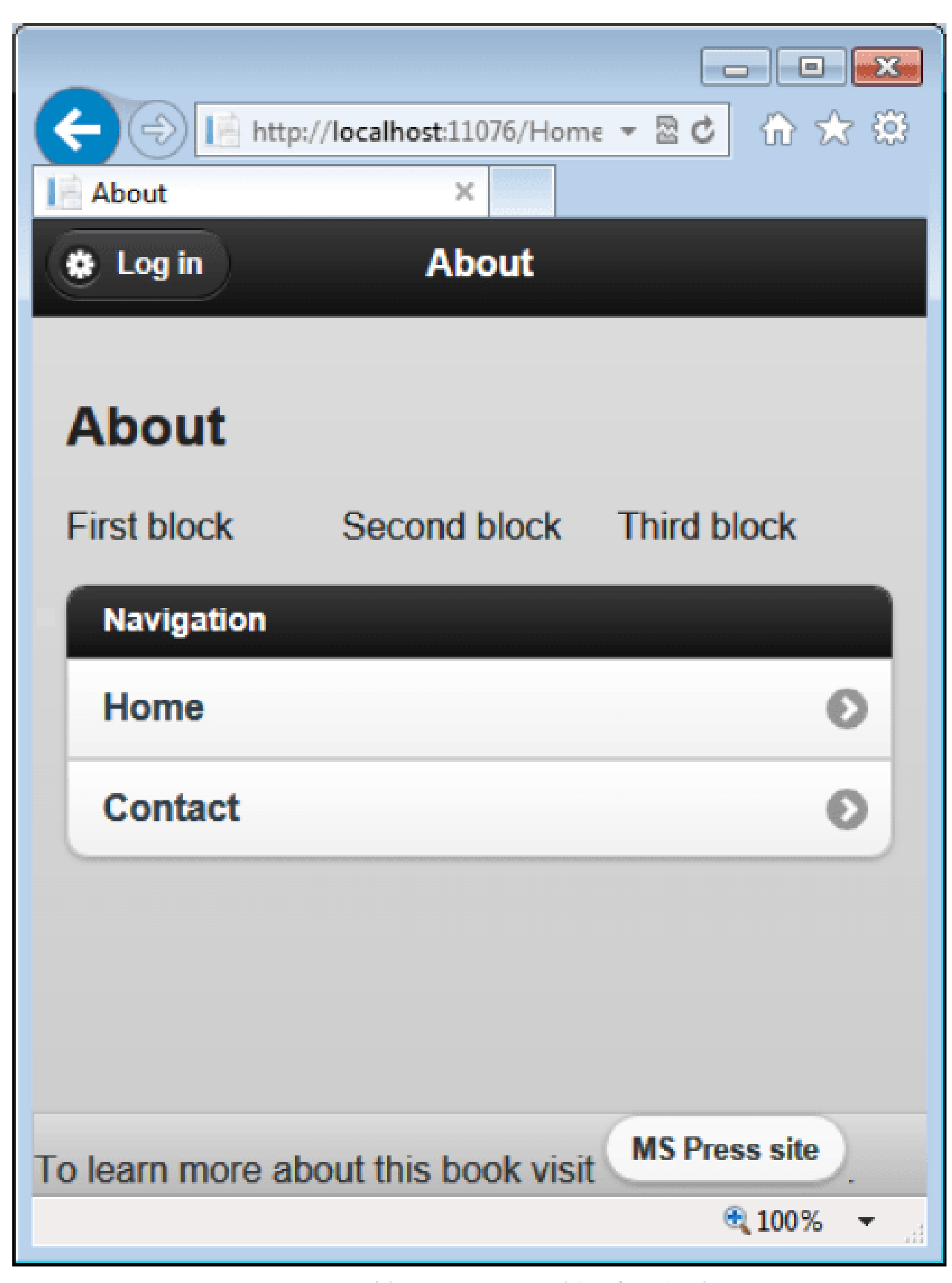


图 12-9　使用 jQM 的流动布局

7. 可折叠面板

可折叠面板也很简单，如以下代码片段所示。

```
<div data-role="collapsible" data-theme="a" data-collapsed="true"
               data-content-theme="e">
    <h3>See navigation options</h3>
    <div>
      <ul>
        ...
      </ul>
    </div>
</div>
```

可折叠角色被分配给容器元素，它使用第一个<Hn>子元素来对面板添加标题并使其他内容可折叠。可以使用 data-collapsed 来决定内容是否要初始化隐藏。可以将样式 data-content-theme 应用到面板内容，并将样式 data-theme 应用到页眉，如图 12-10 所描绘的。

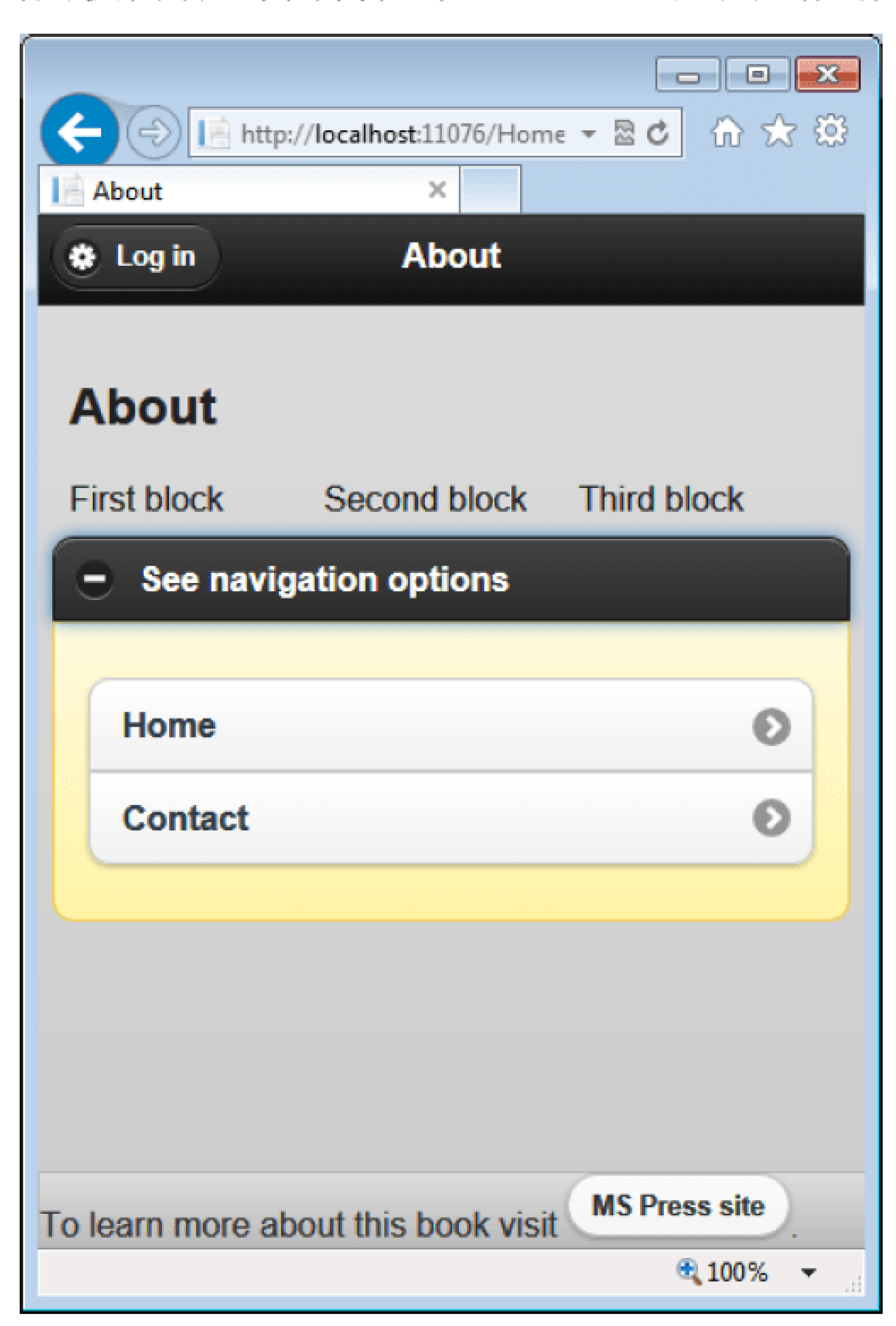


图 12-10　使用 jQM 的可折叠面板

只要用一个最外层容器包裹一系列可折叠面板，你就能获得一个可折叠的部件。

```
<div data-role="collapsible-set">
   <div data-role="collapsible">
```

```
      ...
    </div>
    <div data-role="collapsible">
      ...
    </div>
     ...
</div>
```

通过将样式特性放置在父容器上，所有的面板都能具有相同样式。否则，你就需要对每个面板单独赋予样式。

移动框架

jQM 库是最早出现的用于移动端编程的框架。最初，它的推广很慢，这使得开发人员转而寻找其他选项和供应商提出的新框架。经过几个版本的迭代，jQM 已经变得更加好用并代表了一个可选项。

事实上，即使普通的 JavaScript 也是一个可选项——并且很可能是最简洁快捷的选项——你可能需要四处寻找一些框架来满足用户对美观 UI 和 UX 的要求。如果不使用 jQM，你还能依靠什么呢？

Twitter Bootstrap 是一个有意思的选项。即便该框架并非专为移动端方案而设计，但在需要处理流动布局和集成几个 jQuery 插件以快速实现众多 jQM 特性时，它也能提供极佳的功能。另一方面，Bootstrap 具有完整的敏捷结构。

相比之下，Kendo UI 是一个用于移动设备的非常成熟的框架，它是基于 HTML5 开发的，并且经过全新设计以便为不同类别设备生成高质量和漂亮的 UI。Sencha 也能用于达成这些类似目标。新框架的名单正变得越来越长。

如果对各种框架感到有些困惑，别担心：不只你这样觉得。通常来说，没有哪个框架具有显著的优势。如果发现一个特定框架很适合你，那就持续使用它吧。另一方面，如果使用的框架不太适用，也能很容易找到一个更好的框架等着你下载。

12.1.4　Twitter Bootstrap 概览

如今所有网站都被期望是响应式的——至少要能响应宿主屏幕宽度的变化。作为开发团队的一员，你有两种方式达成此目的：可以简单地从供应商处订购一个 HTML 响应模板(并透明公开地实现细节)，或者可以进行模板的自主开发。在后一种情形中，你会使用哪个框架呢？Bootstrap 大概是你首先会想到的框架，但它不是唯一的一个。

其他同样流行的有效响应式框架有 Foundation(http://foundation.zurb.com)、Skeleton(http://getskeleton.com) 和 Gumby(http://www.gumbyframework.com)。可以在 http://mashable.com/2013/04/26/css-boilerplates-frameworks 找到 CSS 响应式框架和自定义开发方法的快速指南。

Bootstrap 正迅速变成现代 Web 开发的实际标准，特别是现在 Visual Studio 2013 将其集成到了 ASP.NET MVC 应用程序的默认模板中。使用 Bootstrap，可以轻易地安排网页的 UI、让其可响应，以及提供高级导航和图表功能。Bootstrap 本质上由一个 CSS 文件和一个可选的 JavaScript 文件构成。应用到 HTML 页面元素后，Bootstrap 的 CSS 类可以改变当前 DOM 的外观，并且最棒的是代码看起来仍然与普通 HTML 一样。还有一个额外的好处，就是在大多数任务中你甚至都不需要额外的 JavaScript。

1. 设置 Bootstrap

Bootstrap 是以模块来表述的，它最初是作为 LESS 文件集合来开发的。基本上你要让每个组成 Bootstrap 框架的模块具有一个 LESS 文件，这些模块包括：表单、按钮、导航、对话框等。LESS 文件是对普通 CSS 语法的抽象，这使可以声明 CSS 文件最终是什么样的。可以将 LESS 看作是一种编程语言，它能在编译时生成 CSS。为此，使用 LESS，你就能使用变量、函数以及运算符——大大精简了创建较大和复杂 CSS 样式表的过程。

通过对原始 LESS 文件进行修改，开发人员能随意自定义 Bootstrap。如果不适应 LESS 语法，还可通过使用一个配置引擎将轻量级的自定义应用到 Bootstrap。如果不介意 Bootstrap 的内部构成，并且只想要最小化下载量，则可以选取核心 GitHub 存储库中的 CSS 文件，选出你要使用的模块即可。可以从 http://getbootstrap.com 处下载 Bootstrap。

最近，Twitter 发布了 Bootstrap 的 3.0 版本，其中引入了一些新元素，但同时还变更了现有样式的作用。如果有一个 Bootstrap 2.x 站点，在升级之前请查阅一下 http://getbootstrap.com 的迁移指南；一般来说，你可能会遇见一些重大变更。

要开始使用 Bootstrap，只需要将以下代码添加到空 HTML 页面的<head>节即可：

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<link href="bootstrap.min.css" rel="stylesheet" media="screen">
```

viewport 元标签会将浏览器视窗的宽度设置成实际的设备宽度，并将缩放级别设置成普通。<link>元素仅引入了 Bootstrap 样式表的压缩版本。你能够在 http://getbootstrap.com 处下载 Bootstrap CSS 文件，或者从一个已知的内容分发网络(CDN)处链接这些文件。

```
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css">
```

你还需要链接 jQuery 库。这对于制作不带有像下拉菜单或弹出对话框这些额外功能的响应式模板就足够了。如果打算引入一些需要客户端脚本的最高级功能，你还应该添加 Bootstrap.js 文件，可以在下载包中找到该文件。如果选取一个特殊的 Bootstrap 主题，那么还要将该 CSS 文件添加到页面。可以在 http://wrapbootstrap.com/themes 处找到免费的 Bootstrap 主题。

请记住，在 Internet Explorer 的旧版本兼容模式下不支持 Bootstrap。确保页面能在最好的可能呈现模式中浏览的最佳方法是，将以下<meta>标签添加到你的页面中：

```
<meta http-equiv="X-UA-Compatible" content="IE=edge">
```

Bootstrap 的 CSS 类使用了最新的 CSS 特性(比如圆角)，这些特性在旧版本的浏览器上可能无法使用。由 Bootstrap 部分支持的旧版本浏览器的例子就是 Internet Explorer 8。

2. 网格系统

Bootstrap 中的响应性是通过一些断点来达成的。第一个断点被放置在 480 像素，这也是默认的。其他的断点放置在 768 像素(大多数平板电脑)、用于桌面端 992 像素，以及用于大屏幕的超过 1200 像素的断点。Bootstrap 的所有内容通常都是根据以下模式来布局的：

```
container> row > span
```

上述每个单词指的都是一个 Bootstrap 类名称，你通常要将其应用到一个<div>元素。尤其是，container 会管理页面的宽度和内边距。row 样式会确保内容被放置在同一行内，并确保多个行垂直摆放。如果没有 row 元素，则容器中的所有内容都将水平排列，并且当到达屏幕末端时进行换行。最后，span 样式会将行中的内容识别成一个整块。一般来说，如果选择一个不同于 container-row-span 的组合，结果将不可预测。然而，如果能设法得到你确实想要的，就无须在意其是否符合建议的关系。

```
<div class="container">
   <div class="row">
      ...
   </div>
   <div class="row">
      ...
   </div>
</div>
```

一行的内容就是一个流动网格系统的主体，当设备尺寸增大时它会放大到 12 列。你要为一个行单元格使用 col-md-N 样式，其中 N 表示网格中要使用的单元格数量。此外，你还可以使用不同大小的列，这些列要使用像 col-lg-N(大屏幕)、col-sm-N(平板电脑)和 col-xs-N(智能手机)这样的样式。下面是一个例子：

```
<div class="container">
   <div class="row">
       <div class="col-md-3">
       </div>
       <div class="col-md-3">
```

```
        </div>
        <div class="col-md-3">
        </div>
    </div>
</div>
```

上面的代码片段创建了一个带有三个水平排列的<div>元素的块元素，平均占据整个空间。每个<div>中的内容都会在可用空间里居中。

3. 导航栏

通过使用其后跟随有更多指定样式的 nav 基类，可以轻易地将<div>元素转变成导航栏。下面显示了如何创建一个顶级标签菜单。

```
<ul class="nav nav-tabs">
  <li class="active"><a href="#">One</a></li>
  <li><a href="#">Two</a></li>
  <li><a href="#">Three</a></li>
</ul>
```

辅助样式有 nav-tabs、nav-pills、nav-stacked 以及 nav-justified。它们最终会生成图形化效果，比如药丸(间隔均匀的按钮)、垂直堆叠块，以及水平居中块。

navbar 样式会为网站生成响应式导航页眉——有一些与经典工具栏很相似。非常有意思的是，Bootstrap navbars 在小视图中会初始化显示成折叠状态，而在有足够空间且视窗尺寸变大时会水平展开。所有这些行为都是自动并且内置在 Bootstrap CSS 中的。如果还未能说服你的话，请考虑一下，这个样式会让你(不受限制)得到水平菜单的典型行为，它能折叠成几条垂直线并且在宿主窗口大小可容纳的情况下展开。同样，仅通过使用额外的 navbar-fixed-top 或 navbar-fixed-bottom 样式，可以轻易地将 navbar 的位置固定在页面顶部或底部。

有时候，导航栏用于承载应用程序分支结构中页面的逻辑路径。这称为 Bootstrap 中的面包屑导航。以下是你需要的代码：

```
<ol class="breadcrumb">
  <li><a href="#">One</a></li>
  <li class="active"><a href="#">Two</a></li>
  <li>Three</li>
</ol>
```

最后的结果是自动包括链接和分隔符的标记字符串。你不必编写一行脚本或标记就能完成该呈现。

最终，就页面顶部而言，你还可以使用 page-header 样式。它是<h1>标记元素的基本包装，会识别内容中的<small>元素。其良好效果是，子的小文本会自动应用不同的样式以便反

映出下级标题。

4. 用图标和图片修饰 UI

包含 Bootstrap 文件的压缩文件还包括一个装满符号图标的目录，该目录可从 http://glyphicons.com 获得，并且是免许可的。可以在任何需要标记的位置使用这些图标；不需要<img>元素。最通常的情况是，你要将符号包装在<span>标签集中并将其与一些自由文本结合，如下所示：

```
<span class="glyphicon glyphicon-wrench"></span> Tools
```

该标记的效果是在文本“Tools”之前显示扳手符号。可以在<div>元素中使用它，并轻易地将该容器用作按钮样式。值得注意的是，你需要指定两个类：基础 glyphicon 类，其后跟随有标识出你想要的图标的具体类。

在响应式模板中，图片在某种程度上会成为瓶颈，因为它们对于某些屏幕尺寸来说可能太大并且会浪费带宽。此外，当图片太大时，它会截断屏幕并导致难看的 UI。在 Bootstrap 3 中，无须做任何处理你就能轻易地让<img>标签成为响应式的，只要添加 img-responsive 类即可。

```
<img src="..." class="img-responsive">
```

其效果如同添加了以下样式：

```
max-width: 100%;
height: auto;
```

这样一来，图片大小就能很好地缩放成父元素的尺寸。请注意这一技巧在下载时并不适用。它只会确保图片被很好地呈现并适当收缩；不过，下载的图片尺寸总是相同的。

5. 下拉菜单

下拉菜单在计算机 UI 中并非新事物，但它们仅在最近才开始在网页中变得常见。HTML——就连最新的 HTML5 也是如此——没有语法元素能用来在单击按钮时生成下拉菜单。要达成该任务，可以混合使用所有工具：CSS、特设标记以及 jQuery 插件。使用 Bootstrap 3，所有的这些基本细节都会对视图完全隐藏。下面的代码显示了网页中需要的标记，以实现放置一个按钮并将下拉菜单附加到该按钮。

```
<div class="dropdown">
  <button data-toggle="dropdown">Actions</button>
  <ul class="dropdown-menu" role="menu">
    <li role="presentation">
       <a role="menuitem" tabindex="-1" href="#">One</a></li>
    <li role="presentation">
```

```
        <a role="menuitem" tabindex="-1" href="#">Two</a></li>
    <li role="presentation">
        <a role="menuitem" tabindex="-1" href="#">Three</a></li>
    <li role="presentation" class="divider"></li>
    <li role="presentation">
        <a role="menuitem" tabindex="-1" href="#">Four/a></li>
  </ul>
</div>
```

正如你所看到的，所有标记都被包装在样式为 dropdown 类的<div>元素中。dropdown 类是在 Bootstrap CSS 文件中定义的，这个类会为实际 UI 和相关行为做好基础准备。在某种程度上，你能使用一些伪标记来重写上面的全部 HTML 标记，如下所示：

```
<dropdown>
 <trigger>Actions</trigger>
   <dropdown-menu>
       <menuitem href="#">One</menuitem>
       <menuitem href="#">Two</menuitem>
       <menuitem href="#">Three</menuitem>
       <menudivider />
       <menuitem href="#">Four</menuitem>
   </dropdown-menu>
</dropdown>
```

开发人员需要做的就是定义目标 URL 以及其后的代码。它可以是某些内部 HTML 元素的散列标签，或者它也可以是指向某些 JavaScript 代码或外部 URL 的指针。图 12-11 显示了下拉标记的效果。

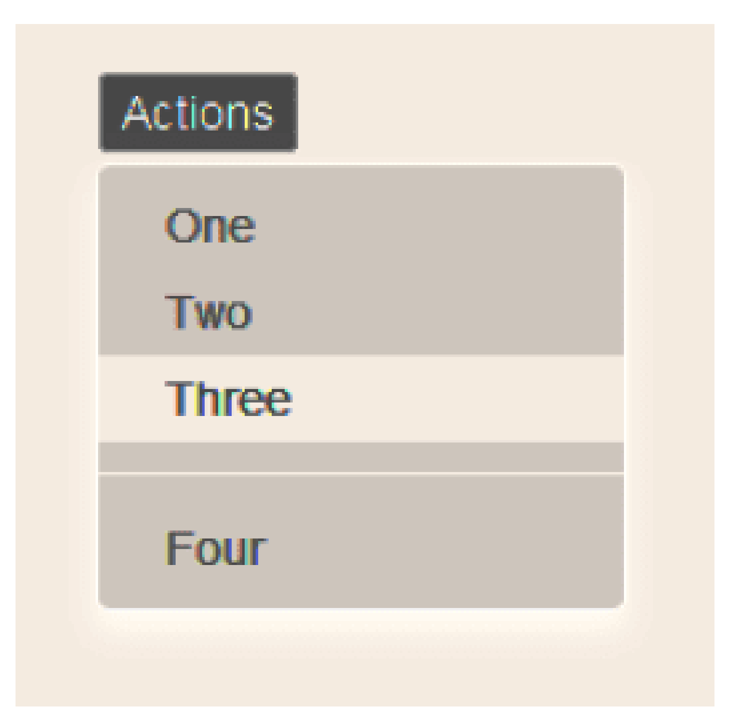


图 12-11　关联到按钮的下拉菜单

下拉菜单有许多其他部分可以自定义。例如，你能在可单击菜单项之间添加一种注释行。你需要做的只是定义一个常规<li>元素并为其分配 dropdown-header 类。

```
<li role="presentation" class="dropdown-header">Special actions</li>
```

可以通过为相应<li>元素添加 disabled 类的样式来禁用一个菜单项，如下所示：

```
<li role="presentation" class="disabled">
<a role="menuitem" tabindex="-1" href="#">Four/a>
</li>
```

要触发该菜单，你需要一个<button>或一个<a>元素，并使用设置为 dropdown 值的 data-toggle 特性对其进行配置。在 Bootstrap 中，你要使用 data-toggle 特性来请求某些元素上的单击行为。该特性的值是一些预定义关键字之一，它主要是指示框架选择一个特定 jQuery 插件来实际执行该苦差事。当你想要通过 dropdownjQuery 插件来完成下拉行为的话，就要使用关键字“dropdown”。

6. 按钮分组

通常，网页需要显示一些在某种程度上相关的按钮。你当然可以单独处理这些按钮并根据你的喜好为其添加样式。不过，几年前，iOS UI 引入了分段按钮的概念；如今，就算不是必要条件，分段按钮也是值得拥有的特性。分段按钮本质上是一组按钮，这组按钮能单独发挥作用但呈现为单个条状按钮。其最好的效果是，该条状按钮中的第一个和最后一个按钮具有圆角，而中间的按钮完全是方形的。在 Bootstrap 中，你要使用以下基于 HTML 的标记：

```
<div class="btn-group">
  <button type="button" class="btn btn-success">Agree</button>
  <button type="button" class="btn btn-default">Not sure</button>
  <button type="button" class="btn btn-danger">Disagree</button>
</div>
```

正如你所看到的，上面的代码真的差不多就是几个单独的按钮而已，每个按钮都有其自己的单击处理程序，这些处理程序要么通过 onclick 特性显式添加，要么通过 jQuery 隐式添加。要具有一个按钮分组，你需要做的就是将按钮清单包装到样式为 btn-group 的<div>元素中。图 12-12 显示这一效果。



图 12-12　iOS 分段按钮的 HTML 对应项

每个按钮的样式都是通过 btn 类单独赋予的；可以通过 btn-xxx 类添加额外的特性来改变背景色：success、danger、warning、info。使用 btn-group-lg、btn-group-sm 或 btn-group-xs 这样的附加类来控制分组中按钮的大小。默认情况下，按钮都是水平堆叠的。要让按钮垂直堆叠，只需要使用 btn-group-vertical 类即可。可以将多个分组包装到一个按钮工具栏中，以便并排放置这些分组：

```
<div class="btn-toolbar">
```

```
  <div class="btn-group">...</div>
  <div class="btn-group">...</div>
  <div class="btn-group">...</div>
</div>
```

最后，可以嵌套按钮分组。当一个分组要作为下拉列表呈现时，嵌套就特别有用。下面是一个示例：

```
<div class="btn-group">
  <button type="button" class="btn btn-default">One</button>
  <button type="button" class="btn btn-default">Two</button>
  <div class="btn-group">
    <button type="button" class="btn dropdown-toggle" data-toggle="dropdown">
      Numbers
      <span class="caret"></span>
    </button>
    <ul class="dropdown-menu">
      <li><a href="#">1</a></li>
      <li><a href="#">2</a></li>
      <li><a href="#">3</a></li>
    </ul>
  </div>
</div>
```

该分组的前两个项是普通按钮，然后是一个嵌套分组。该按钮分组由一个附加了下拉菜单的按钮组成。应当注意的是，该按钮的特征是具有一个插入符号分段，直观地表示有更多可用的选项。被分配到按钮中<span>元素的 caret 类会呈现一个向下的箭头，类似于经典的 Windows 下拉箭头。

注意：

Bootstrap 所具有的特性远远不止此处所讨论的。要了解更多特性，请访问 http://getbootstrap.com。

12.2 为已有站点添加移动功能

规划移动站点的最重要任务是选择用例。不过，这并不是说开发完整站点或其他类型的应用程序时用例选择不重要。只是因为移动应用程序和站点在结构上是围绕一些(精心选择的)用例来构建的。有时候，即使你从已有桌面站点中选取用例，你用来为移动用户实现它的方法也可能需要大量修改——可能是一个不同的 UI，也可能是一个不同的工作流。在面临为移动用户提供完全不同视图的情况时，RWD 并不总是像你需要它的那样强有力。

所以，不似很多人认为的那样，通常的结果是你需要一组用于全部浏览器的页面，然后将多组页面用于你要支持的每种移动设备。现实情况是，桌面端正变成特殊情况。你要如何设法让这多组页面看起来像单个站点呢？在第 13 章中，我会讨论由 ASP.NET MVC 提供的根据请求设备将用户路由到特定视图的原生工具。

然而有时候，更简单的做法是构建一个用于移动用户的从头开始设计的独立网站，然后通过使用 HTTP 模块和 cookies 将这两个站点连接起来。

12.2.1　将用户路由到正确的站点

因为你现在拥有两个单独的站点，所以需要一种自动化机制根据请求设备的性能将用户切换到合适的站点。如果宿主名称属于桌面站点且请求浏览器被检测为桌面浏览器，则一切将如预期运行。否则，用户将看见一个目标页面，该页面会提示用户他正试图用移动设备访问桌面站点。此处用户有机会可以保存其偏好以便将来出现类似的情形时使用。其偏好会被存储到 cookie 并在下一次检查。

1. 路由算法

如果请求涉及移动站点的 URL 而用户似乎是在使用桌面浏览器，可以考虑显示另一个着陆页面，而不是简单让该请求顺利通过。最后，如果一个请求是从移动设备发出且目标是移动站点时，它将如预期一样被处理；也就是说，通过检查设备性能并确定最合适的视图即可。图 12-13 显示了该算法图表。

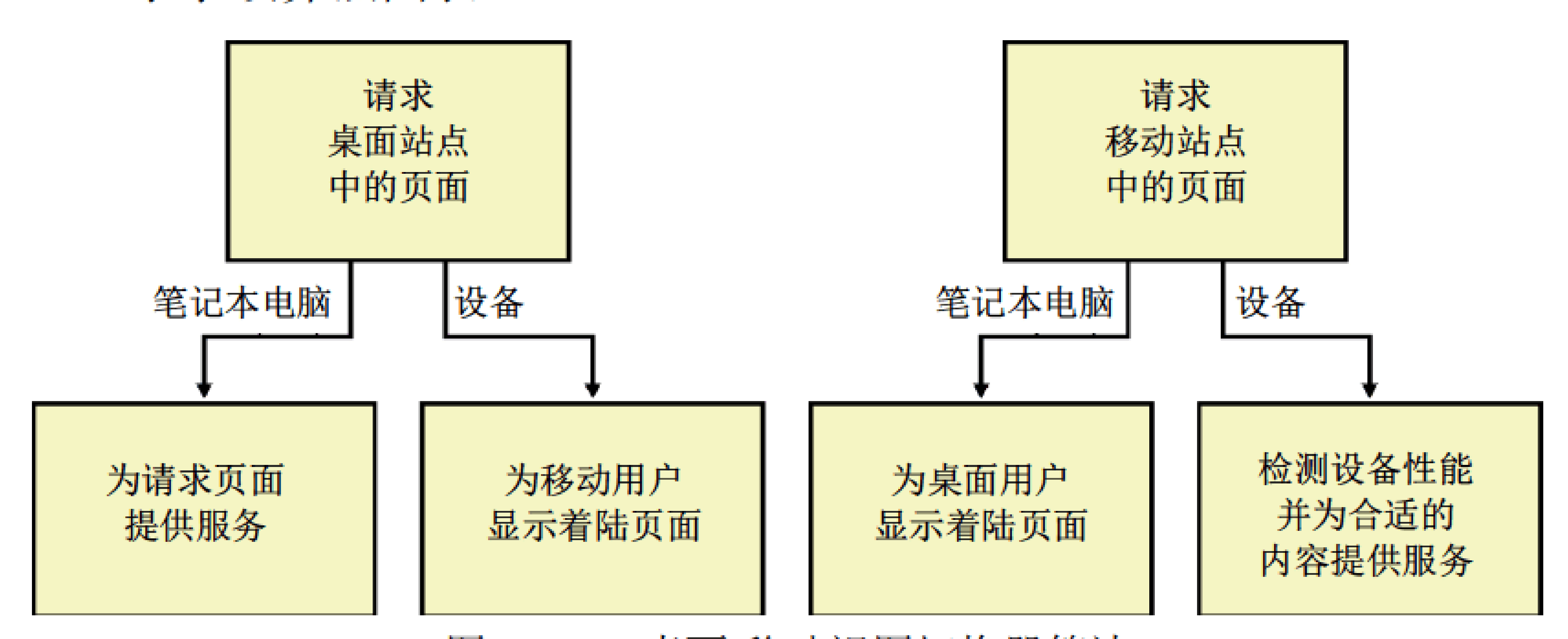


图 12-13　桌面/移动视图切换器算法

注意：

常常有一种错误，就是认为桌面和移动页面之间存在一对一的关系。这种情况有可能发生，但它不应被认为是通常会出现的情形。这里所说的“页面对应关系”，指的是这两个应用程序都能为相同的 URL 提供服务；我并没暗示每个页面都会实际提供的任何相关内容。

你要如何实现图 12-13 中描绘的算法呢？

2. 实现路由算法

在 ASP.NET 中，实现这一路由算法的原生工具是，使用一个作用于两个站点的 HTTP 模块并捕获 BeginRequest 事件。该模块会使用普通的重定向，或者可能的话，使用 URL 重写来酌情变更目标页面。

以下是在桌面站点中实现前述算法的一些代码：

```
public class MobileRouter : IHttpModule
{
   private const String FullSiteModeCookie = "FullSiteMode";
   public void Dispose()
   {
   }
   public void Init(HttpApplication context)
   {
      context.BeginRequest += OnBeginRequest;
   }

   private static void OnBeginRequest(Object sender, EventArgs e)
   {
      var app = sender as HttpApplication;
      if (app == null)
         throw new ArgumentNullException("sender");

      var isMobileDevice = IsRequestingBrowserMobile(app);

      // Mobile on desktop site, but FULL-SITE flag on the query string
      if (isMobileDevice && HasFullSiteFlag(app))
      {
         app.Response.AppendCookie(new HttpCookie(FullSiteModeCookie));
         return;
      }

      // Mobile on desktop site, but FULL-SITE cookie
      if (isMobileDevice && HasFullSiteCookie(app))
         return;

      // Mobile on desktop site => landing page
      if (isMobileDevice)
         ToMobileLandingPage(app);
   }
```

```
        #region Helpers
        private static Boolean IsRequestingBrowserMobile(HttpApplication app)
        {
            return app.Context.Request.IsMobileDevice();
        }

        private static Boolean HasFullSiteFlag(HttpApplication app)
        {
            var fullSiteFlag = app.Context.Request.QueryString["m"];
            if (!String.IsNullOrEmpty(fullSiteFlag))
                return String.Equals(fullSiteFlag, "f", StringComparison.
                                     InvariantCultureIgnoreCase);
            return false;
        }

        private static Boolean HasFullSiteCookie(HttpApplication app)
        {
            var cookie = app.Context.Request.Cookies[FullSiteModeCookie];
            return cookie != null;
        }

        private static void ToMobileLandingPage(HttpApplication app)
        {
            var landingPage = ConfigurationManager.AppSettings
                              ["MobileLandingPage"];
            if (!String.IsNullOrEmpty(landingPage))
                app.Context.Response.Redirect(landingPage);
        }
        #endregion
    }
```

在其被安装到桌面站点后，该 HTTP 模块会捕获每一次请求并检查请求浏览器。如果浏览器运行在移动设备中，则该模块会重定向到指定的目标页面。该目标页面应该是一个面向移动优化的页面，其实质是提供两个链接，分别链接到桌面站点的首页和移动站点的首页。图 12-14 显示了在旧的 Android 2.2 设备上浏览的示例目标页面。

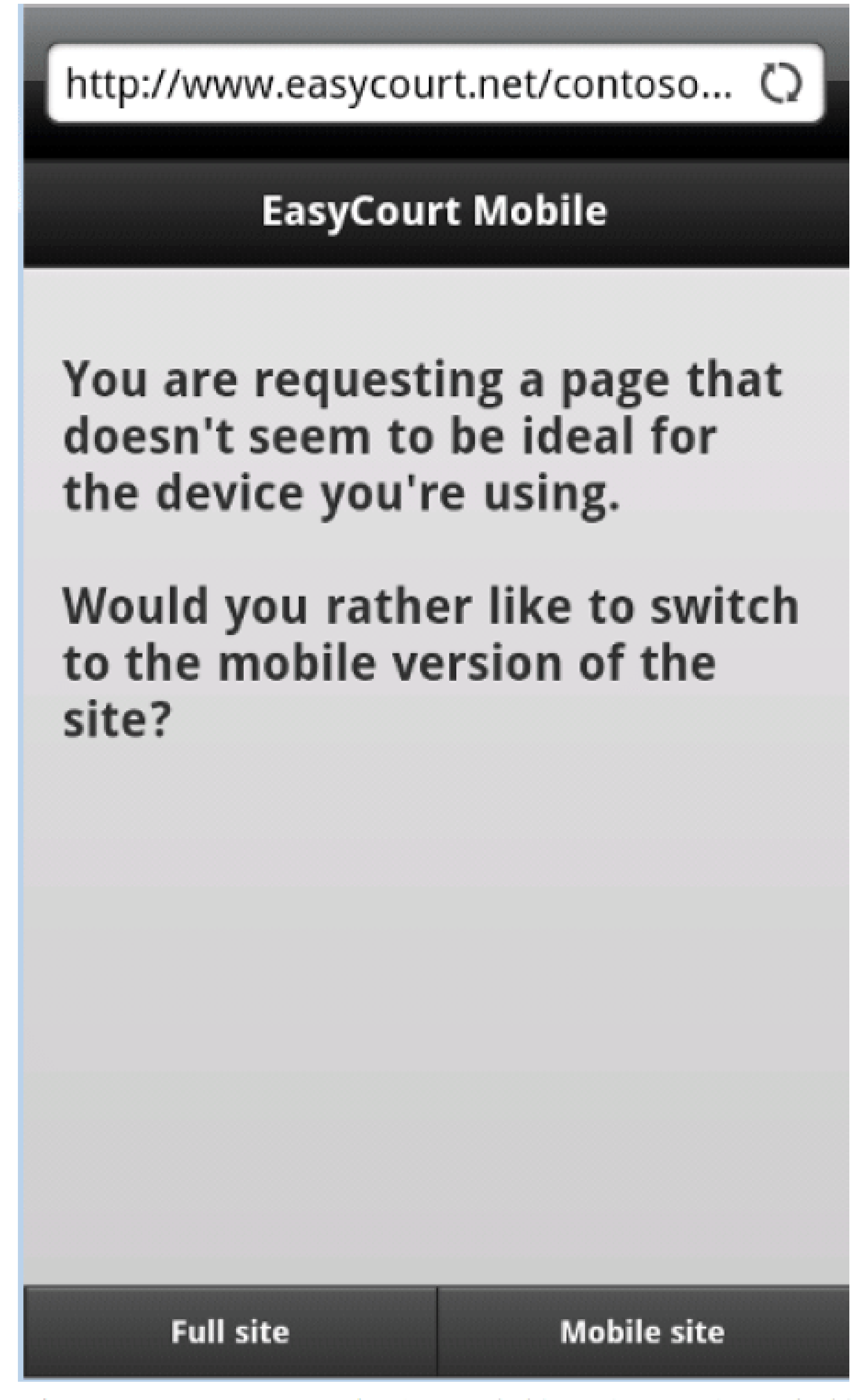


图 12-14　在 Android 2.2 设备上浏览的示例移动站点的着陆页面

3. 跟踪所选路由

如果用户坚持要浏览完整站点，你就不能仅仅重定向到普通的主页面了。就其本质而言，该 HTTP 模块将解析这一新请求并再次重定向到移动着陆页面。在该着陆页面中，可以简单地添加一个指定查询字符串参数，以便由该 HTTP 模块在连续的请求上检测。以下是产生图 12-14 所示页面的实际链接：

```
<a href="http://.../?m=f">Full site</a>
```

你要负责定义该查询字符串语法；在这个例子中，m 代表模式，而 f 代表完整。不过，该任务还没有完成。此时，用户导航到了站点首页。那其他请求又如何呢？实际上，那些请求将由该 HTTP 模块解析。通过添加 cookie，你就能为该 HTTP 模块提供额外的信息，这些信息与有意从移动设备提交请求到桌面站点有关。

用户要如何切换回移动站点呢？理想情况下，具有关联移动站点的所有桌面站点都应该提供一个显而易见的链接来切换到移动版本(反之在移动设备上浏览完整站点时也应如此)。否则，在 cookie 失效或被清除之前，用户将无法在完整站点或移动站点之间进行选择。要清

除 cookies，用户要使用移动浏览器的设置页面。

4. 微调配置文件

你要在哪里放置着陆页面呢？它位于桌面站点还是移动站点上？通常，这不重要；但是，如果将其放置在移动站点上，你就真的能实现一种方案，在其中部署一个具有全部所需路由逻辑的移动站点，而无须使用已有桌面站点的代码库。

不过，桌面站点需要在其配置中做一些修改。具体来说，你要编辑桌面站点的 web.config 文件，并在 Bin 文件夹中部署一个带有该 HTTP 模块的库。不要对源代码进行任何修改。以下是将路由 HTTP 模块添加到桌面站点的配置脚本：

```
<system.webServer>
    <modules>
       <add name="MobileRouter" type="..." />
    </modules>
    ...
</system.webServer>
```

请记住，欢迎页面应该总是可见的，并且绝不应该对其进行身份验证。根据你部署移动站点的方式——单独的根站点/应用程序或者子应用程序/目录——你可能需要微调移动站点的 web.config 文件，以停用该 HTTP 模块。如果移动站点是一个单独的应用程序，那么它需要其自己的完全配置有该 HTTP 模块的 web.config 文件。不过，如果移动站点在桌面站点中是作为子目录承载的，则它就会继承父站点(桌面站点)的配置设定，包括该 HTTP 模块。要加速处理请求，你可能会想要在移动站点中禁用该 HTTP 模块。

以下是移动站点的 web.config 文件中需要的配置脚本。该脚本会清除移动站点所需的 HTTP 模块列表。

```
<system.webServer>
    <modules>
       <clear />
    </modules>
    ...
</system.webServer>
```

此外，你需要指示父应用程序/站点显式停止默认的设置继承链。以下是所需的代码：

```
<location path="." inheritInChildApplications="false">
<system.webServer>
    <modules>
       <add name="MobileRouter" type="..." />
    </modules>
    ...
```

```
</system.webServer>
</location>
```

另外请注意，当移动站点是一个子应用程序/目录时，它会继承一串无须重复(比如，连接字符串和成员资格提供程序)的设置(那些继承性未被关闭的节)。

注意：

该示例基于默认的互联网信息服务(IIS)7.5 配置，也就是基于集成管道模式。如果使用的是经典管道模式，就应该在 system.web/httpModules 节上而非 system.webServer/modules 上操作。

12.2.2 从移动端到设备

直到两三年前，网站都是用于桌面端浏览器或移动端浏览器的。然而，“移动端”一词已经日益模糊了。移动端指的是智能手机还是平板电脑，又或者两者都是？另外，迷你平板或大屏智能手机呢？普通手机又是不是移动端呢？智能电视呢？重点是，你真的需要开始考虑多种设备的问题，并决定你的 RWD 是否能达成你的目标或者是否需要寻求其他更有效的设备检测方法。针对这一点来说，将单个“移动端”站点添加到已有的桌面端站点的方法看起来就像是临时解决方案或者说仅仅是补丁式解决方案。

合理的情形是，你应该至少将“移动端”一词分成两类：智能手机和平板电脑。另外还建议增加其他类别，比如大屏设备(如智能电视)和传统旧款手机，但在所有应用程序中都并非强制必须的。

如果选择一个与已有桌面站点密切关联的移动站点，这是否意味着你需要具有两个或三个额外的站点呢？当对多种类别设备的支持迫在眉睫时，你应该研究多设备设计。从概念上讲，它在某些方面与 RWD 相同；尤其是，它推进了单个后端和调用单个 URL 的观念。然而，从更深层次来看，多设备设计与 RWD 不同，因为它推进了不同类别设备的不同视图的使用。视图不只是一个不同的 CSS 文件那么简单，它还包括不同的标记和脚本。

多设备并不意味着你要为每种设备创建一个不同的 UI；而且它不意味着你必须负责识别指定设备。设备的类别(智能手机、平板电脑、大屏设备)差不多与 RWD 中的断点相同。需要进行特性检测，但在服务器端需要一些特设工具的帮助，比如设备描述库(DDR)。第 13 章将进一步介绍这方面的详细内容。

12.3 本章小结

如今我看见的挑战是，我们要如何找到一种将移动需求与桌面需求结合在一起的编程范式。许多工作的重点都是让移动端的外观和表现与桌面端相似。我不清楚这是否是达成移动开发的正确方式。

打个比方，让我们假设五年之内最终将出现比今天更多的强大设备；不过，移动设备的基本特征并不会发生变化，它将仍然比笔记本电脑小且性能弱一些。此外，就商业智慧而言，还需要处理长尾效应的问题。为了仅支持高端设备，你一路放弃了多少设备呢(可能将其留给你的竞争对手来支持了)?

尽管有一些可行的解决方案来让网站对设备友好，但主要路由要穿过为不同类别设备的不同视图提供服务的架构。有时候，不同的视图通过修改 CSS 文件即可获得；而有时这样做还不够，还需要不同的标记和脚本。在前一种情况中，RWD 和客户端特性检测是非常好的选择。否则，就需要实现服务器端特性检测，但很可能是通过使用智能工具及非手工编写代码的方式来进行。

在这一章，我们了解了让网页响应不同浏览器窗口大小所需的基本知识。我们了解了 HTML5 这一响应式网页设计基础，以及 Twitter Bootstrap 这一最常用的创建网站响应式体验的方法。此外，我们还介绍了 jQuery Mobile，这是作为移动视图的一个示例 JavaScript 框架来研究的。

第 13 章

构建用于多种设备的站点

未来属于那些今天有准备的人。

——*Malcolm X*

我预计在几年之内，那些不能很好地被主流设备所使用的网站将面临流量显著下降的情况。我有意使用了“主流设备”一词而非更具体的像平板电脑或智能手机这样的名词，只是鉴于环境变化非常快这一原因：没人能明确知道几年内会出现哪些设备并且被广泛使用。

不过，除了这些预测之外，有一件事可以肯定：传统手机将成为历史。它们会变得与恐龙一样古老。智能手机的定义也在发生变化，仅仅两年前的设备就和最新一代版本的性能和功能显著不同了。在这种背景下，没人愿意继续进行 Web Forms 网站的开发，因为它需要在设备可视区中伸缩，并需要用户来放大和缩小进而阅读与单击链接。

如何创建既美观又对各类设备高度可用的网站呢？

有两种主流想法。一些人声称应该一直专注于设备实际公开的特性。这称为特性检测，这一方式是客户端所固有的，并且要使用层叠样式表(CSS)/JavaScript 框架来决定哪一个层更适合于请求页面。另外，与客户端特性检测相关的还有响应式网页设计(RWD)，一种利用高级 CSS 功能推送流体布局以便将内容碎片顺序加载到可用空间中的流行设计方法。

另一种极端的情况是，一些人更愿意在服务器端进行处理。他们喜欢识别出设备浏览器发送的用户代理字符串并静态解析出已检测到的设备的一些功能。然后，根据这一信息，服务器端解决方案就能够智能地提供为请求设备量身定做的特设标记。

RWD 是一种好方式；服务器端也是一种好方式。可以使用这两种方式来完成解决方案；有时候这两种方式混合使用功能更加强大。客户端解决方案支持者的呼喊声肯定比服务器端方式支持者的大。这已经吸引了越来越多的人，这些人都认可这一观点，即对比在不可维护的混乱怪异的用户代理字符串中进行搜索，特性检测更加智能和快速。

总的来说，我相信 RWD 的优点很好理解，而每天层出不穷的额外框架能够减少 RWD 的缺点。但是，我也相信服务器端解决方案的优点并没有被真正理解，而其所谓的缺点不过是以前已经完全过去的软件时代的遗留物而已——也就是桌面应用程序与浏览器之争的时代。

在这一章中，我的目标是呈现服务器端特性检测方式的优缺点，以及可能用于实现这一方式的额外(并且非免费)工具。

13.1　理解 ASP.NET MVC 中的显示模式

多年来一个网页一直就只是一个网页而已。例如，如果一个用户请求 default.aspx，他就只会得到为 default.aspx 计算的内容，这通常与请求浏览器的能力无关。只要请求浏览器都融入桌面浏览器这个相同类别，就差不多是有效的。不同桌面浏览器版本之间在呈现上可能会有轻微(而恼人的)不同，但通过在 HTML 中使用一点 jQuery 和一些分支代码，我们已经围绕它工作多年了。HTML5 的到来添加了超越标记的另一种程度的差异：功能。值得庆幸的是，像 Modernizr 这样的库为我们解了围，这些库有助于一些功能的检测以及提供填充层的基础架构。

然而，支持多种设备(并且不只是桌面浏览器)是一个不太一样的问题。

多种设备意味着请求浏览器可以是明显不同的类型，因为它们可能是安装在智能手机、平板电脑、智能电视、超大屏幕设备或超小屏幕设备上的。这不仅仅是让相同内容以最佳可行方式适应于呈现的问题；它已经变成了要同时让内容和行为适应于完全不同种类的设备以及尽可能实现不同的用例集的问题。

在多设备方案中，前面提到的 default.aspx 页面会分解成许多页面，比如 default.smartphone.aspx、default.tablet.aspx 等。因此每个逻辑页面可能会根据用户定义规则以数种方式显示。

显示模式组成的 ASP.NET MVC 中的基础架构，能够有效地处理单个逻辑页面的多种视图。

13.1.1　分离移动视图和桌面视图

即便我坚定地认为软件中不会存在神奇之处，但还是不得不说，当我进行以下步骤以及碰到 ASP.NET MVC 显示模式时我对自己产生了严重怀疑。

1. 移动视图的内置支持

在 ASP.NET MVC 中，现在你能体验到一种有意思的特性，它表面上运行得就像是纯粹魔法造成的结果。假定你有一个带有普通简单 Index 方法的 HomeController 类，就像如下所示的这样：

```
public class HomeController : Controller
{
   public ActionResult Index()
   {
      var model = ProcessRequestAndGetData();
      return View(model);
```

```
        }
    }
```

第 2 章“ASP.NET MVC 视图”指出，位于项目中 Views/Home 文件夹内的名为 index.cshtml 的文件会为浏览器提供 HTML 以便呈现。

现在你要将一个视图文件添加到 Views/Home 文件夹并将其命名为 index.mobile.cshtml。可以给予该文件任何你喜欢的内容；只要确保这些内容不同于之前提到的 index.cshtml 即可。现在，启动示例站点并用常规桌面浏览器(例如，微软 IE)和移动端浏览器访问它。可以使用 Windows Phone 模拟器或 Opera 模拟器。令人惊讶的是，你得到的 home/index URL 的视图是在 index.mobile.cshtml 文件中编码的移动视图。

注意：

总体来说，进行功能测试的最简单且无需太多繁琐操作的方式是，在 IE 中按下 F12 键以显示 Developer Tools 窗口。在该窗口，可以设置一个匹配移动端设备的模拟用户代理。如果不确定要输入的内容，这里有匹配运行 iPhone OS 6 的 iPhone 的建议：Mozilla/5.0 (iPhone; CPU iPhone OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko)。

这里到底发生了什么？这一切又是怎么发生的呢？

2. 移动视图的默认配置

让我们逐步对 ASP.NET MVC 的代码进行了解。从控制器方法中调用视图方法时，在使用 CSHTML 视图的所有情形中，代码执行流会到达 RazorViewEngine 类。在 ASP.NET MVC 中，所有预定义的视图引擎都继承自同一个类：VirtualPathProviderViewEngine。这个类具有 DisplayModeProvider 类型的一个名为 DisplayModeProvider 的受保护属性。视图引擎会接收在控制器级别设置的该视图的名称：该名称可以是“index”，也可以是空字符串，如同之前的示例。如果没有提供视图名称，视图引擎会假设其为操作的名称。

在 VirtualPathProviderViewEngine 基类中，WebFormsViewEngine 与 RazorViewEngine 都继承自该基类，在视图名称解析期间，视图引擎会查询 DisplayModeProvider 对象以检查是否有注册的显示模式能够应用到请求视图。

每个显示模式都是以一个后缀字符串为特征的，比如“mobile”。如果该后缀字符串与已有的视图名称匹配成功，则原始的视图名称就会被修改以指向表示匹配项的 CSHTML 文件。所以，打个比方，可能会发生“index”变成“index.mobile”的情况，如果项目中存在这样一个视图文件的话。

事实证明，默认的 DisplayModeProvider 包含两个预定义显示模式：默认与移动。默认显示模式的特征是空字符串；移动显示模式的特征是“mobile”字符串。这些字符串基本上表明了附加到视图名称的后缀。这也就是 index.mobile.cshtml 这一文件名称的来历了。

13.1.2　选择显示模式的规则

最终，显示模式提供程序指的是一个内部组件，该组件会决定视图引擎需要呈现的实际视图。它会接收在控制器级别设置的视图名称并决定是否用指定设备视图的名称对其进行替换，如果该指定设备视图存在的话。有一个问题：驱动这一选择的逻辑是什么？默认情况下，显示模式提供程序能在常规桌面视图与移动视图之间进行选择。但是，是什么来确定最佳的选择呢？

1. 对显示模式命名

在 ASP.NET MVC 中，显示模式是由名称为 DefaultDisplayMode 的类实例来表示的。这里有一个移动显示模式的定义：

```
var mobileMode = new DefaultDisplayMode("mobile") {
    ContextCondition = context => context.GetOverriddenBrowser().
                                  IsMobileDevice
};
```

显示模式类是围绕两条主要信息来构建的：后缀名称与匹配规则。在之前的代码段中，使用“mobile”这一后缀创建了一个新的显示模式类，并向 ContextCondition 属性分配了一条匹配规则。

显示模式的名称可以是你希望你的站点支持的能唯一标识特定视图的任意字符串。按照惯例，空字符串标识的是默认视图——在大多数情况下是桌面视图，但这不是必须的。默认情况下，移动视图能识别出任何其他的请求浏览器，通过使用一些内部 ASP.NET 逻辑可以将这些浏览器的用户代理字符串匹配到一种移动设备。

值得注意的是，移动显示模式管理的设备属于非桌面设备种类，包括老式的传统手机、最新的智能手机、平板电脑、小型平板等。还有，移动显示模式并不会区分操作系统。对此，我们完全无法区分 iPhone 设备与安卓设备，并且对运行安卓 2.1 设备的处理方式与对运行安卓 5.0 设备的处理方式也是相同的。

这正是你要采用某些深层自定义进行干预的领域，以便量身定做将提供给各种设备的标记。

2. 匹配规则

DefaultDisplayMode 类具有一个名为 ContextCondition 的属性，你要使用该属性来检查请求的上下文并将其匹配到一个受支持的显示模式。ContextCondition 属性是一个委托，其定义如下所示：

```
Func<HttpContextBase, Boolean>
```

该委托的目的是分析当前请求的 HTTP 上下文，并返回一个布尔值以回答后面这个问题：这一显示模式应该被用于服务当前请求吗？如何得到该布尔值响应完全取决于匹配规则的实现。

在 ASP.NET MVC 中，移动显示模式的默认实现会解析随请求而来的用户代理字符串，并试图找出能够将请求发起者标记为移动设备的已知关键字。

注意：

归根结底，事实证明 DefaultDisplayMode 类的名称存在一些误导。它仅仅是定义一个显示模式的类而已。在我看来，该名称中的“Default”前缀并不合适。

13.1.3　添加自定义显示模式

显示模式是一个真正的强有力的特征，但你只有在清除默认配置并创建你自己的配置后才能充分利用它。首先，要对该 API 熟悉，让我们先看看如何列出现有的显示模式。

1. 列出当前的显示模式

你基本上不需要在代码中执行该任务，但处于纯粹的兴趣，我还是建议你尝试一下。下面是读取和显示当前可用模式的代码：

```
<ul>
    @{
        foreach(var d in DisplayModeProvider.Instance.Modes)
        {
            <li>@(String.IsNullOrEmpty(d.DisplayModeId) ?"default" :
                                                        d.DisplayModeId)</li>
        }
    }
</ul>
```

你要使用 Instance 静态成员来访问 DisplayModeProvider 类的单例实例并逐一查阅 Modes 属性。图 13-1 显示了通过 IE 开发者工具栏分别选择移动和桌面用户代理字符串的示例页面。

现代网站需要比移动/桌面两种选择更多的选项来用于选择视图。最有可能的情况是，你会想要区分平板电脑、智能手机、老式手机甚至智能电视。有时候，一个简单的 CSS 处理就足以给予你所需的结果。这就是 RWD 能够完成的任务。然而还有一些时候，你需要通过设备提供的用户代理字符串来进行设备的服务器端检测。

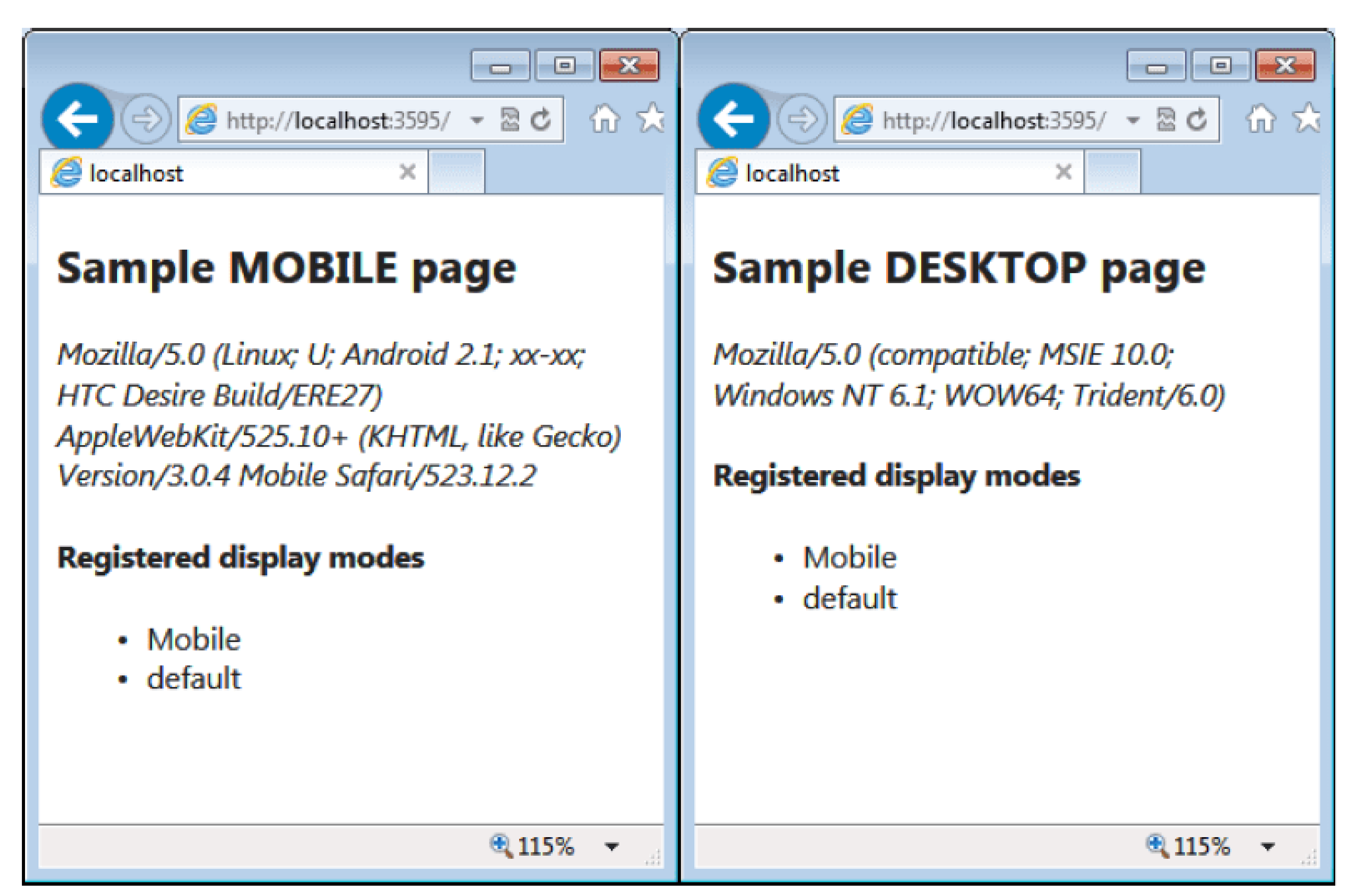


图 13-1　在指定页面的桌面与移动视图之间进行选择的 ASP.NET MVC 应用程序示例

2. 超越默认配置

即使普通的桌面/移动这两种选择能够满足你站点的需求，你也应该注意默认移动上下文条件背后的逻辑是脆弱的这一事实。这有很大的可能性对 iPhone 与黑莓设备起作用，但甚至不可能对 Windows Phone 与安卓设备起作用，更不用说更老旧与更简单的设备了。之前你所看到的被引用的 IsMobileDevice 方法确实能够根据它在.browser 文件中找到的信息检测出用户代理字符串，.browser 文件是随着 ASP.NET 一起安装的，如图 13-2 所示。

该模型显然可扩展，并且任何时候你都可以添加更多的信息；但是对.browser 文件进行编写并不简单，而且测试、检查和进一步扩展数据库的担子也完全压在你的肩上。

如果将图 13-2 的内容与你在你的互联网信息服务(IIS)计算机上具有的相同文件夹中的内容进行对比，你可能会注意到一些不同之处。尤其是，图中的文件夹包括一个很大(18MB)的浏览器文件——实际上是一个 XML 文件——名为 mobile.browser。该文件来自于一个古老的、已停止的微软项目，其中包含了截至 2013 年夏天出现的合理数量的设备(参见 mdbf.codeplex.com)。所有后来出现的设备与浏览器都不能被正确检测。

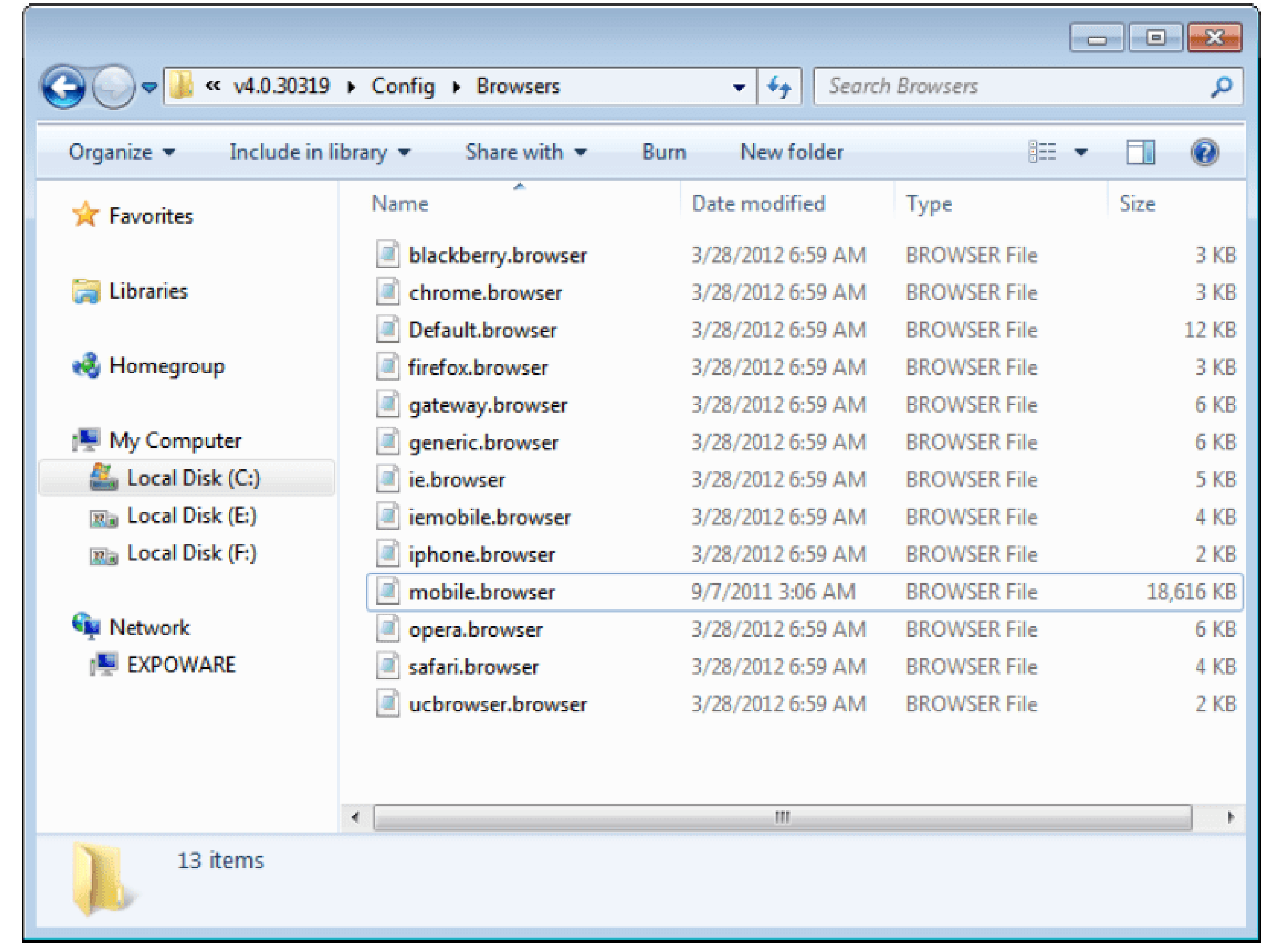


图 13-2　在服务器计算机上全新安装的 ASP.NET 所提供的开箱即用的浏览器配置文件列表

奇怪的是，ASP.NET MVC 基础架构并没有做出相应的修改以便更确切地处理这方面的内容。归根结底，显示模式是基础架构很棒的一部分，但需要你承担一些工作来配置并且需要额外的工具来有效地执行视图路由工作。更为重要的是，要让 ASP.NET MVC 站点对移动设备友好，你需要选取一个外部框架来帮助进行设备检测(稍后我将继续介绍这方面的内容)。

3. 定义自定义显示模式

你需要使用显示模式来赋予你站点同一个 URL 的多个视图。具体来说，这主要是指为每种设备或你关注的设备种类定义一个显示模式。例如，可以创建一个 iphone 显示模式。同样，也可以创建一个平板电脑显示模式。通常而言，显示模式对于创建特定于移动设备的视图极其有用，但它们仅仅是创建特定视图的工具而已，这些特定视图会在应用到 HTTP 上下文的布尔条件判定值为 true 时得到应用。下面是一些定义了几个自定义显示模式并替换了默认配置的代码：

```
var modeTablet = new DefaultDisplayMode("tablet")
{
   ContextCondition = (c => IsTablet(c.Request))
};
var modeDesktop = new DefaultDisplayMode("desktop")
{
   ContextCondition = (c => return true)
```

```
};
displayModes.Clear();
displayModes.Add(modeTablet);
displayModes.Add(modeDesktop);
```

该代码要从 Application_Start 处运行，它会丢弃默认显示模式并定义两个新模式：平板电脑和桌面。首先添加的是平板电脑模式并且它会被首先检查。实际上，找出合适显示模式的内部逻辑会在第一次匹配时停止。如果 HTTP 请求不匹配平板电脑，那么它接着会由带有为桌面设备优化的视图的默认项进行处理。

请注意这一点，因为桌面模式是与一个特定的非空字符串(desktop)相关的，所以每个视图都需要具有适当的后缀以供识别。换句话说，index.cshtml 视图文件上的配置并不会被使用。相反，你需要使用名为 index.desktop.cshtml 和 index.tablet.cshtml 的视图文件。

13.2　WURFL 数据库介绍

现在关键的问题就变成了后面这个：你如何才能可靠地确定某个指定请求是否来自于一台平板电脑？

简单的答案是，这一切都与嗅探用户代理字符串有关。但是等等：嗅探用户代理字符串不正是将开发人员引向充满喜悦地拥抱 RWD 以及客户端特征探测的一种噩梦吗？其次，举个例子，客户端特征探测难道不足以检测设备的操作系统及其他许多特定于设备的功能吗？

服务器端特征探测不仅仅是一长串在无穷无尽的 switch 语句中的分支——每个 switch 语句对应一种可能和已知的代理。更为重要的是，你不会想要亲自对用户代理嗅探进行编码。如果这样做了，将很可能无法采用现有成熟的方案，使你自己陷入疯狂开发和维护的艰难境地。要让服务器端特征探测不成为负担并且高效，你需要一个专业的用户代理嗅探工具：设备描述存储库(DDR)框架。DDR 是像预言者一样运行的组件，它会揭示所有(已知的)与正浏览页面的移动端浏览器有关的实情，以便你能够明智地决定作为响应来提供的标记。

WURFL(参见 http://wurfl.sourceforge.net/)是当前使用最广泛的 DDR，但它肯定不是唯一的一个。像 Facebook 和谷歌这样的大企业都将 WURFL 用于其移动站点。而且它对于开源项目还是免费的，并且还有部分免费的云版本。不过，对于商业使用，你很可能需要购买授权。更多信息请参阅 http://www.scientiamobile.com。

注意：

不管选择的是 WURFL 还是其他产品，比如 51 degrees.mobi，关键点是你不应在构建自己的嗅探用户代理字符串的解决方案上投入精力。这不仅是因为它会耗费大量时间，还因为鉴于移动端浏览器的大量种类，它很可能会产生一个脆弱的解决方案。

13.2.1　存储库的结构

WURFL DDR 包括一个压缩后约 1.5 MB 大小的 XML 文件，在扩展后也不会超过 20 MB。可以从 http://wurfl.sourceforge.net 下载该文件最新的快照。

注意：

下载 WURFL 数据库时，必须明确同意一些条款和条件。大体来说，你会被授权使用未修改的 WURFL 文件并仅通过由 ScientiaMobile 提供的一个标准 WURFL API 来使用该文件。对于 ASP.NET，可以找到一个 NuGet 包来下载和安装最新的 WURFL API 和数据库。

1. XML 总体架构

WURFL 数据文件由一个<device>元素的简单列表构成。这里是该数据库的总体框架：

```
<devices>
  <device id="..." user_agent="..." fall_back="...">
    <group id="...">
      <capability name="..." value="..." />
      ...
    </group>
    ...
  <device>
  ...
</devices>
```

id 特性会通过名称唯一标识一种设备。user_agent 特性表示要匹配的特定用户代理字符串。

关键的特性是 fall_back，通过名称将其引用到其他<device>元素。fall_back 特性会表明当前设备将从哪种设备处继承缺失的性能。换句话说，每一个设备节都仅描述了当前设备与其父设备之间的差异。所有设备都会直接或间接引用一个通用根设备，该根设备会确保所有支持的性能总是具有一个默认值并且在查询期间不会抛出异常。WURFL 支持多个通用根设备，每种可识别的设备类别一个，这些可识别设备类别包括：移动电话、平板电脑、智能电视以及未来可能的更多设备。

2. 性能分组

每种设备都是与一个性能列表关联的。一个性能会被描述成名称/值对，其中值往往被认为是一个字符串。这意味着 WURFL API 将总是返回普通字符串类型的性能值，而不会尝试将该值匹配成特定的类型，比如布尔型或整型。

之所以这样做是为了特别对待 API 的扩展性以及性能，而非其他。实际上，有不少性能

会从值的枚举中获取值。例如，pointing_method 性能表明了链接是如何在设备上激活的。可能的值有触控笔、手柄、触摸屏、点击轮、或空字符串。所有这些选项都能够毫无问题地作为 Java 和.NET 语言中的枚举类型来表示。不过在这种情形下，为添加新的可行指针方法而对数据文件所做的任何扩展也都需要对 API 进行修改，并可能会中断现有的应用程序。

为了保持可管理性，就要对性能进行分组。表 13-1 列出了当前可识别的分组。不过，分组在 API 中毫无作用，从这个意义上讲，无须对信息分组以检索性能的值。

表 13-1　WURFL 中浏览器和设备性能的分组

分　组	描　述
ajax	尽管是这个名称，但这一分组还定义了超出普通 Ajax 编程的性能。它不仅向你表明 Ajax 是否受支持，还向你表明 DOM 与 CSS 操作是否被允许以及能够处理地理位置信息
bearer	与网络特性相关的性能，比如支持无线电广播、Wi-Fi、虚拟专用网(VPN)以及最大访问带宽
cache	与嵌入式浏览器的缓存配置相关的性能
chtml_ui	与压缩式 HTML 标记相关的性能
chips	与通过安装在设备上的额外芯片提供的可用特性相关的性能，比如 FM 收音机和近场通信(NFC)装置
css	与 CSS 特性相关的性能，比如图片拼合、边框、圆角以及倾斜
display	与屏幕大小(以像素与毫米计)和方向相关的性能
drm	与支持一些 DRM 标准相关的布尔值性能
flash_lite	与对 Flash 应用程序类型和版本内置支持有关的性能
html_ui	与用 HTML MIME 类型提供的内容相关的性能。该分组包括与视口、HTML5 画布、内嵌图片以及首选文档类型描述(DTD)有关的属性
image_format	与一些图片格式支持相关的布尔值性能
j2me	告知开发人员哪些 Java 特性可用于 J2ME 运行时中的 midlets 的性能，这些特性包括定位、屏幕大小、套接字、图片、多媒体以及更多
markup	与各种受支持的标记类型相关的布尔值性能，其中包括 XHTML、无线标记语言(WML)以及 HTML
mms	与 MMS 相关的性能，比如图片支持、视频以及最大帧率
object_download	与可下载对象相关的性能，比如视频剪辑、图片、墙纸、屏幕保护以及来电铃声
pdf	与原生支持 PDF 内容相关的性能
playback	与受支持的视频格式和从网站下载的内容解码器相关的性能
product_info	与设备相关的性能，比如品牌和型号名称、是移动设备、电话还是平板电脑、操作系统、键盘以及浏览器

(续表)

分　　组	描　　述
rss	与原生支持 RSS 源相关的性能
security	与 HTTPS 支持和 IMEI 可见性相关的性能
sound_format	与各种不同音频格式相关的布尔值性能
smarttv	与智能电视相关的性能
sms	与 SMS、EMS(富文本 SMS)以及电话铃声相关的性能，包括像诺基亚和松下这样一些供应商的特性
storage	与设备可以管理的页面大小相关的性能
streaming	与支持的视频格式和来自网站内容流的编码相关的性能
transcoding	这些性能的目标是识别来自代码转换器的请求——代码转换器是一款能够用作网关并隐藏真实设备信息的软件。这些性能是提供给你以便在需要以特殊方式处理请求的时候使用的
wap_push	这些性能的目标是探测有效无线应用协议(WAP)的特性
wml_ui	与 WML 标记相关的性能
xhtml_ui	与 XHTML 标记相关的性能

下面列出了揭示通用设备 CSS 性能的一段代码摘要——该通用设备指的就是 WURFL 根设备：

```
<group id="css">
  <capability name="css_gradient" value="none" />
  <capability name="css_border_image" value="none" />
  <capability name="css_rounded_corners" value="none" />
  <capability name="css_spriting" value="false" />
  <capability name="css_supports_width_as_percentage" value="true" />
</group>
```

作为对照，以下示例显示了一种通用安卓设备的相同分组：

```
<group id="css">
   <capability name="css_border_image" value="webkit"/>
   <capability name="css_rounded_corners" value="webkit"/>
   <capability name="css_spriting" value="true"/>
   <capability name="css_supports_width_as_percentage" value="true"/>
</group>
```

如你所见，一些属性可以重写。

3. WURFL 补丁文件

WURFL 存储库带有两个或多个文件(任意命名的)：一个是代表存储库自身的 XML 文件(通常命名为 wurfl.xml)；其他的都是补丁文件，通常使用 xxx_patch.xml 这种模式来命名。补丁文件都是可选的。

注意：

WURFL API 是基于配置模块的，通过它可以指向存储库和可选的补丁文件。在 ASP.NET 中，你要么可以通过连贯接口以编程方式达成该任务，要么通过 web.config 文件中的自定义节来达成。

使用补丁文件，你就可以改变默认存储库内的一些性能而无须物理调整原始文件(而且物理调整原始文件将损坏授权，即使你是在合法获取的副本上进行的)。换句话说，补丁文件提供了一种重写 WURFL 数据库内某些内容的方法。在解析 WURFL 文件时如果发现了补丁，那么补丁内容就会被导入以构建一个修改后的存储库版本。这里有一段补丁文件的摘要，该补丁文件是为 Firefox 10 添加支持的(以防止该版本浏览器不受最新更新的存储库支持)：

```
<device user_agent="Firefox" fall_back="generic_web_browser" id="firefox">
    <group id="product_info">
        <capability name="brand_name" value="firefox" />
    </group>
</device>
<device user_agent="Mozilla/5.0 (Windows NT 5.1; rv:10.0) Gecko/20100101
                   Firefox/10.0"
       fall_back="firefox" id="firefox_10_0">
   <group id="product_info">
       <capability name="model_name" value="10.0"/>
   </group>
</device>
```

为何你会想要使用补丁文件呢？

总的来说，使用补丁文件的主要原因是，出于你自己的理由为某些性能分配不同的值。例如，假设你为平板电脑设备定制了一个网站。接着，你碰到了一种特殊设备，其屏幕尺寸足够大，能够完全容纳你设计的平板电脑的用户界面。不过遗憾的是，WURFL 仍然会将该特殊设备认作与平板电脑不同的其他设备。然后你就要创建一个新补丁文件(或编辑已有的补丁文件)并重写 is_tablet 性能，以便支持该特殊设备的用户代理。下面是示例代码：

```
<device user_agent="your nice tablet device that WURFL doesn't consider a
                   tablet"
       fall_back="generic_mobile" id="mytablet">
    <group id="product_info">
```

```
            <capability name="is_tablet" value="true" />
        </group>
    </device>
```

补丁文件起作用的另一场景是当你需要一个在 WURFL 中不受原生支持的性能时，这要么是因为它对于你的应用程序来说太过特殊，要么是因为之前根本没人考虑过它。最后，当发现有一些错误数据存在于原始 WURFL 数据库中时，补丁文件也能够起到帮助作用。关于补丁文件的更多信息以及示例，请访问 http://wurfl.sourceforge.net/patchfile.php。

13.2.2　基础 WURFL 性能

让我们仔细看看一些 WURFL 性能，确切地了解可以通过 WURFL 获得的对被送达内容的控制水平。下面是对超过 600 个可用性能的一个小范围选择。可以在 http://wurfl.sourceforge.net/help_doc.php 找到有关 WURFL 功能的完整文档。

1. 识别当前设备

关于请求设备，你需要了解的最常见类型的信息是其身份。表 13-2 列出了一些非常有用的性能，这些性能描述了被用来执行当前请求的设备。该表显示了性能的名称、其 WURFL 分组、其描述以及其可能的值。

表 13-2　与设备相关的性能

性　　能	WURFL 分组	值	描　　述
is_wireless_device	product_info	真/假	设备是无线的
is_tablet	product_info	真/假	设备是平板电脑
is_smarttv	smarttv	真/假	设备是智能电视
device_os device_os_version	product_info	字符串	当前设备的名称和版本(比如 Android 2.2)
resolution_width resolution_height	display	整数	以像素为单位的屏幕宽度和高度
max_image_width	display	整数	以像素为单位的图片能在设备上被浏览的最大宽度
can_assign_phone_number	product_info	真/假	设备可以与手机号码关联。这用于区分仅使用 SIM 来浏览 Web 的设备
pointing_method	product_info	手柄、触控笔、触摸屏、点击轮、“”	该方法用于选择链接。注意空字符串表示的是设备上的四向导航，它具有上下左右四个按钮来导航链接

(续表)

性　　能	WURFL 分组	值	描　　述
brand_name model_name marketing_name	product_info	字符串	品牌(比如 HTC)、型号名称(比如 HTC A8181)、以及设备的营销名称(比如 HTC 渴望)

其中一些属性可以使你非常精确地对设备进行分类。例如，可以检查传入的请求是否来自承载在无线设备上的浏览器。is_wireless_device 为任何能匹配到移动设备的用户代理字符串返回 true，这些设备包括手机、PDA、平板电脑(不包括笔记本电脑和 AppleTV 这样的智能电视)。如果要做的只是检测移动设备，那么此属性正是你所需要的。要进行更详尽的分析，你还可以检查会在 iPad 上返回 true 的 is_tablet ，以及在手机(可以具有关联的电话号码)上返回 true 的 can_assign_phone_number，但在 iPod 上就不行。另一个类似的性能是 has_cellular_radio(在 bearer 组中)：此性能会表明设备是否可以安装 SIM，不管其用途是什么。可以在 iPad 上使用 SIM 卡，但不能在像 iPod Touch 这样的设备上使用 SIM 卡。

如果需要区分 iOS 和 Android 或 Windows Phone 设备，可以使用 device_os 和 device_os_version 性能。如果需要知道准确的设备信息(制造商和产品名称)，则可以使用 model_name 和 brand_name。

已知的实际屏幕大小是由 resolution_width 和 resolution_height 返回的。最后，有关触摸性能的信息，当值等于 touchscreen 时，会由 pointing_method 功能返回。

注意：

有关操作系统的任何信息都暗含在用户代理中。当在设备上安装一个操作系统的新版本时，它应该也会更新浏览器，且浏览器应当在发送的 us-er-agent 字符串中反映该操作系统的版本。然而，这只是预期的工作方式。例如，在旧设备上遇到系统是 Android 2.2 版本、但浏览器仍然反映的是 2.1 版本的情况是不足为奇的。

2. 提供特定于浏览器的内容

表 13-3 列出了一些性能，可以帮助你微调提供给浏览器的标记。WURFL 处处充满了能够对所提供标记进行微调的性能。下面的性能代表了你需要在服务器上使用不同标记模板以生成视图的场景。

表 13-3　提供特设内容的性能

性　　能	WURFL 分组	值	描　　述
viewport_supported	html_ui	真/假	浏览器支持<viewport>元标签
image_inlining	html_ui	真/假	浏览器能显示通过数据统一资源标识符(URI)结构嵌入的图片
full_flash_support	flash_lite	真/假	浏览器完全支持 Flash
cookie_support	xhtml_ui	真/假	浏览器支持 cookies
preferred_markup	markup	字符串	想要提供给浏览器的标记类型
png, jpg、gif、tiff、greyscale	image_format	真/假	浏览器能显示指定类型的图片

一些移动浏览器会假定它们能够呈现每一个页面，所以它们会把页面缩小为实际的屏幕大小，让用户以比较便利的方式通过缩放来浏览页面的某一部分。HTML <viewport>元特性的引入使开发人员能够指示虚拟屏幕——视口——该有的尺寸。但是，<viewport>元标签并非是标准的，在发出之前检查一下会更安全。viewport_supported 性能与其名称表示的意思是一样的。

浏览器将图片视为独立的资源并触发一个额外的请求来下载它们(如果未在本地缓存的话)。对于移动设备，HTTP 请求的执行成本远比在桌面浏览器中大得多，因此任何技术都是受到欢迎的，只要它能减少完成页面所需的 HTTP 请求的数量。常用的技术包括在 HTML 页面中嵌入小的图片作为 base64 编码的文本。这种技术被称为图片内联，在一些老的设备上不受支持，只有对能够最小化下载的设备类别才更重要。

有了 image_inlining 功能，可以提前知道请求浏览器是否能够正确显示以这种方式嵌入的图片。如果这样的检查失败了，可能会遇到的最严重问题是，图片被浏览器用于缺失图片的特定占位符所取代。图片内联是不可能从浏览器内部通过某些智能的 JavaScript 代码来检查的功能之一。

在移动网络中，只有两种类型的标记语言是明显相似的：

- XHTML MP　这是浏览器可以极快速解析和呈现的移动优化标记格式。另外，通过查看多用途互联网邮件扩展(MIME)的类型，任何浏览器都可以合理地识别出该页面是一个移动页面。可惜，该标记语言不像 HTML 那么强大，对 DOM 操作、CSS 和 JavaScript 的支持也并不高级——至少不是以跨浏览器的方式。
- HTML/视窗　这就是普通的 HTML 标记外加<viewport>元标签。HTML/视窗实际上是由 iPhone 和 Safari 为移动设备引入的。与用于完整网页的 MIME 类型相当，苹果公司添加了<viewport>元标签来充当浏览器识别由移动端所展示的页面的一个线索。

在 WURFL 中，preferred_markup 性能会指示哪种类型的标记适合指定的浏览器。如果该性能返回 html_wi_oma_xhtmlmp_1_0，那么你就要提供 XHTML MP 标记；如果返回值是

html_web_4_0，那么你最好提供普通的 HTML 并使用<viewport>元标签将页面标记为移动。

提前知晓页面是用于移动端的有助于浏览器安排最佳的呈现，避免了缩小的页面以及需要放大才能进行有效交互的必要。

13.3　在 ASP.NET MVC 显示模式下使用 WURFL

如前所述，通过一个 NuGet 包，WURFL 库即可用于 ASP.NET。NuGet 包会下载一个二进制格式的示例 WURFL 数据库文件。请记住，你通过该软件包获得的 WURFL 数据库文件可能不是最新的文件。要获取最新的 WURFL 数据库的公共块，请访问 http://wurfl.sourceforge.net。

13.3.1　配置 WURFL 框架

使用 WURFL 需要一些简单的步骤，比如获得二进制文件、通过项目引用这些文件，以及初始化 WURFL 运行时。

1. 安装 NuGet 包

将 WURFL 添加到 ASP.NET 项目的最简单方法是借助 NuGet。该软件包的名称是 WURFL_Official_API(参见图 13-3)。另外，该软件包在 App_Data 文件夹下安装 WURFL 数据库(一个压缩文件)。你不需要解压该文件；API 在解压的时候也会对其进行处理。

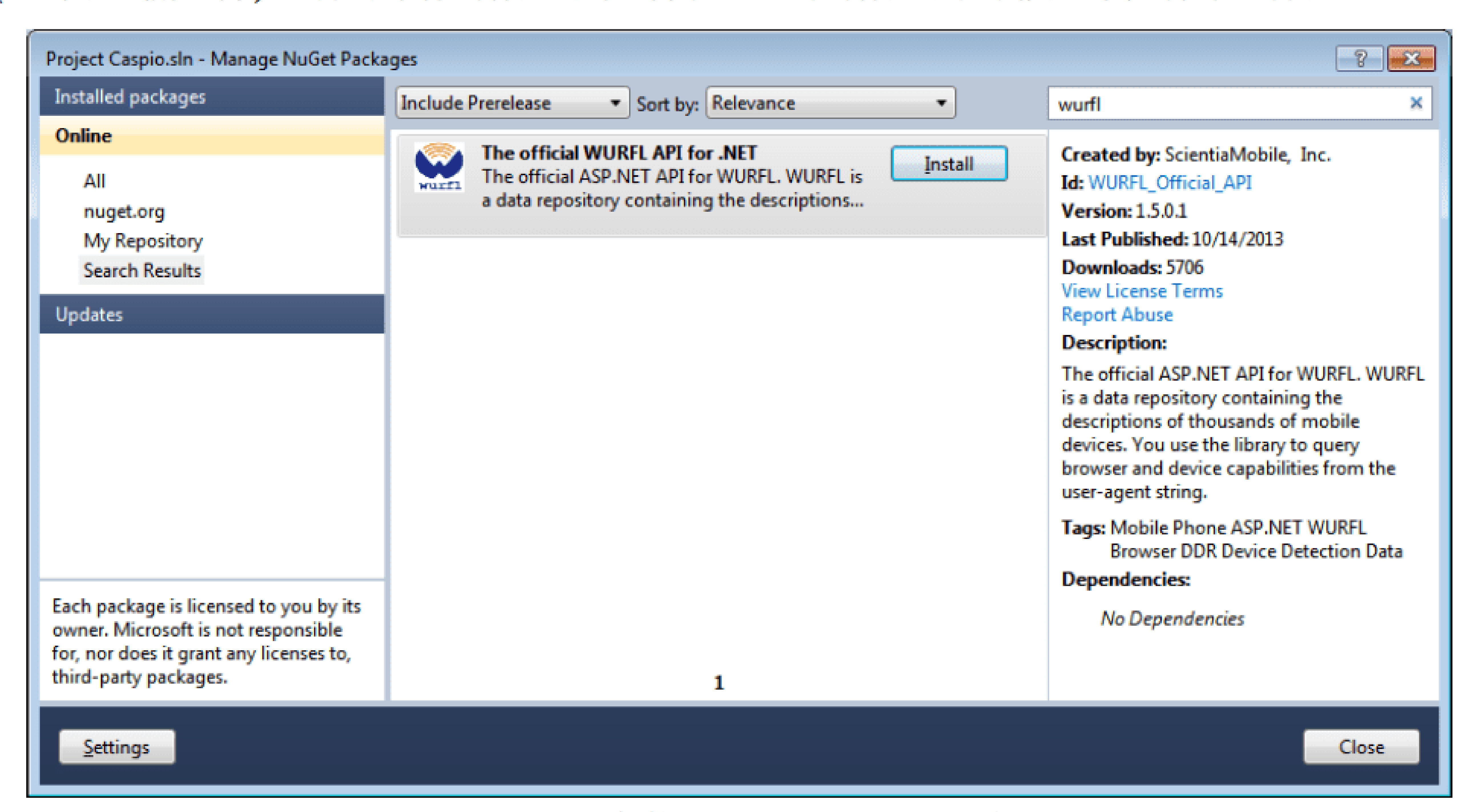


图 13-3　安装 WURFL API NuGet 包

NuGet 包还会复制一些文档和一个内省工具(本质上是一个 HTTP 处理程序)，当将其放置在线上时，能够有助于以调试为目的的对 WURFL 数据库的远程查询。可以将它从项目中

安全移除，因为它在 API 的常规功能中不发挥任何作用。NuGet 包还给 web.config 文件带来了一些变化。具体来说，NuGet 包添加了以下新的节:

```
<configSections>
   <section name="wurfl" requirePermission="false"
          type="WURFL.Aspnet.Extensions.Config.WURFLConfigurationSection,
              Wurfl.Aspnet.Extensions, Version=1.5 ..." />
</configSections>
```

它还提供了一些自身要用到的内容。

```
<wurfl mode="Accuracy">
  <mainFile path="~/App_Data/wurfl-latest.zip" />
</wurfl>
```

wurfl 节不是严格必需的，你从 NuGet 获得的配置也只代表了配置 WURFL 的一种可能方式而已。wurfl 节主要是让你设置 WURFL 数据库的位置，以及应用可选补丁文件的位置。可以使用 ASP.NET 根运算符(~)来表示虚拟路径。不过，也可以通过使用本地 API 以编程方式在配置以外设置源和补丁文件。

2. 引用设备数据库

可以通过使用一个配置对象以两种方式将 WURFL 库指向源数据库。如果想要通过 web.config 文件指定数据库路径，可以使用 ApplicationConfigurer 对象。请将下面的行添加到 Application_Start:

```
WURFLManagerBuilder.Build(new ApplicationConfigurer());
```

ApplicationConfigurer 类会从 web.config 文件读取路径信息。相反，如果想要以编程方式指定 WURFL 数据库的位置以及可选补丁文件的位置，就使用 InMemoryConfigurer 对象。

```
var configurer = new InMemoryConfigurer()
             .MainFile(wurflDataFile)
             .PatchFile(yourWurflPatchFile1)
             .PatchFile(yourWurflPatchFile2);
```

如果需要，你还可以通过实现 IWURFLConfigurer 接口来创建自己的配置类。该接口相当简单，只依赖一个方法，如下所示:

```
public interface IWURFLConfigurer
{
   Configuration Build();
}
```

Configuration 对象也是公共 WURFL API 的一部分。欲知详情，可以阅读 http://wurfl.sourceforge.net/docs/dotnet 的文档或下载其源代码。

3. 初始化 WURFL 运行时

WURFL 库的工作机制是加载全局内存中的数据库内容以及按照调用方的要求提供响应。所有返回的响应都会被无限期缓存，因此不会在后续的请求中重新计算，除非应用程序重新启动。数据库的初始加载发生在应用程序启动时；如果在某一点你替换了服务器上的 WURFL 数据库，则还需要重启应用程序以使更改生效。

WURFL 库的接入点是 WURFL 管理器对象，它是由上述 WURFLManagerBuilder 类型上的 Build 方法返回的。该生成器主要做一件事情：定位 WURFL 数据库并将其内容加载到内存数据结构。因为 WURFL 数据库实质上是一个 XML 文件，因此加载的过程包括读取整个文档并将其解析成合适的零碎部分。

最终包含已解析的 WURFL 数据的内存数据结构会充当 WURFL 管理器私有的内部缓存。这个数据结构会占用大部分 WURFL 所需的运行时内存。最后，当你持有了 WURFL 管理器的一个实例时，你就持有了实际的 WURFL 数据和用来读取它的工具。WURFL 数据在 ASP.NET 应用程序的上下文中应该被视为全局的。

现在，问题变成了你如何引用从应用程序任意位置启动时所创建的 WURFL 管理器的单实例？在内部，WURFL 管理器的构建类似于一个单例。由生成器所创建(然后返回)的 WURFL 管理器类的实例会被分配给名为 Instance 的生成器类的一个公共静态成员。在所有你需要进行 WURFL 查询的位置，都需要引用该管理器，如下所示：

```
var deviceInfo = WURFLManagerBuilder.Instance.GetDeviceForRequest(userAgent);
```

请注意，WURFL 管理器绝不能在内部重置。如果在应用程序启动时正确地对其进行了初始化，那么管理器就不可能变成 null；当然，除非你的代码中有一个管理器变量被分配了一个潜在的 null 引用的路径。

13.3.2　检测设备性能

现在让我们看看处理用户代理字符串的必要条件，并尽可能地了解调用设备。

1.处理 HTTP 请求

WURFL 管理器对象有一个名为 GetDeviceForRequest 的方法，它是你用来了解调用设备的主要工具。

```
var deviceInfo = WURFLManagerBuilder.Instance. GetDeviceForRequest(userAgent);
```

GetDeviceForRequest 方法具有几个重载，可以在用户代理字符串中传递调用，就像

ASP.NET HttpRequest 对象甚或特定于 WURFL 的 WURFLRequest 对象一样。接受 WURFLRequest 类型的重载是用于模拟测试的最简单灵活的一个重载，因为你还可以用它来列出 HTTP 标头和用户代理字符串。

GetDeviceForRequest 方法会返回一个实现公共 IDevice 接口的内部对象。在该接口上的所有成员当中，最重要和最常用的是 GetCapability。该方法会将性能的名称作为一个字符串接收，并将用于标识设备的值作为字符串返回。下面的代码显示了如何获知当前设备的操作系统(如果有的话)：

```
String os = deviceInfo.GetCapability("device_os");
```

对 GetCapability 的每次调用首先都会检查内部缓存，然后对内存中的数据库进行从头至尾的读取。所有计算结果都会缓存以供进一步使用。出于明显的性能原因，WURFL 管理器会保留其自己的私有缓存。可以测量一下，WURFL 库的启动通常需要几秒钟的时间(只会在调用 Application_Start 时发生一次)，而每个请求的提供时间只有几毫秒，并且常常是突如其来的。WURFL 内部缓存使用了最近极少使用的(Least-Recently-Used，LRU)算法，它会自动填充可能已经淘汰的设备信息。

由于这个原因，可以反复调用 GetCapability 以读取多个属性。不过，你还有一个 GetAllCapabilities 方法可用，并且只有在.NET API 的 1.5 版本中你才可以依靠 API 筛选器来限制出于性能原因由 API 管理的能力。

重要提示：

WURFL 功能总是返回字符串值。将所有返回的字符串转变成更易于管理的可用数据类型，比如整数或布尔值，这是开发人员要负责处理的任务。你应该仔细地检查文档，以确保在转换成基本.NET 类型的过程中你没有遗漏某些可能的响应字符串。

2. 虚拟性能

在 ASP.NET 网站中使用 WURFL 的一个常见场景是为不同类别设备提供不同的标记。要实现这一目标，你要能够根据设备的用户代理字符串快速地对其归类，并确定它是智能手机、平板电脑还是传统手机。WURFL 还提供了一些用于这一目的的虚拟性能。

对虚拟性能的处理方式与常规性能相同；它们之所以被称为虚拟是因为它们不具体引用设备的单个和特定特性。典型的例子是名为 is_smartphone 的性能。该性能会指出用户代理是否能够与智能手机设备相关联。从与用户代理关联的性能识别智能手机的算法深植于 WURFL 的内部，主要是检查操作系统的版本、触摸功能和屏幕宽度。表 13-4 显示了 WURFL 虚拟性能的完整列表。

表 13-4　WURFL 虚拟性能

虚 拟 性 能	描　　述
is_android	如果设备运行的是 Android 的任意版本，则返回 True
is_ios	如果设备运行的是 iOS 的任意版本，则返回 True
is_windows_phone	如果设备运行的是 Windows Phone 6.5 或更高版本，则返回 True。注意这不包括 Windows Mobile 或 Windows CE
is_app	如果请求来自本地应用程序，则返回 True。这通常指的是请求来自 WebView 组件或本地应用程序进行 REST API 调用的情况
is_full_desktop	如果请求设备具有完整的桌面体验，则返回 True
is_largescreen	如果请求设备的屏幕具有高分辨率(宽度和高度超过 480 像素)，则返回 True
is_mobile	如果设备是像手机、平板电脑、媒体播放器、便携式游戏机等等这样的移动设备，则返回 True
is_robot	如果请求来自机器人、爬网器或其他一些自动 HTTP 客户端，则返回 True
is_smartphone	如果设备是智能手机，则返回 True。从内部看，该匹配器会检查操作系统、屏幕宽度、指示方法以及其他一些性能
is_touchscreen	如果主要的指示方法是触摸屏，则返回 True
is_wml_preferred	如果请求设备应该由 WML 标记提供服务，则返回 True
is_xhtmlmp_preferred	如果请求设备应该由 XHTML-MP 标记提供服务，则返回 True
is_html_preferred	如果请求设备应该由 HTML 标记提供服务，则返回 True
advertised_device_os	返回请求设备的操作系统。这对于移动设备和桌面设备都有效(例如：“Windows”、“Mac OS X”)
advertised_device_os_version	返回请求设备的操作系统版本。这对于移动设备和桌面设备都有效(例如：“XP”、“10.2.1”)
advertised_browser	返回请求设备的浏览器名称。这对于移动设备和桌面设备都有效(例如：“Internet Explorer”、“Chrome”)

要使用虚拟性能，可以使用一个稍微不同的 API，如下所示：

```
String response = deviceInfo.GetVirtualCapability("is_smartphone");
```

至少，可以通过一个补丁文件将任何用户代理重写成一个智能手机。

3. 精度与性能对比

WURFL 引擎有两种工作模式将用户代理与一组性能进行匹配：精度和性能。你通常要

通过配置器来设置工作模式。如果从 web.config 文件获取配置，就可以通过 wurfl 节点的 mode 特性来指定工作模式，如下所示：

```
<wurfl mode="Accuracy">
  <mainFile path="~/App_Data/wurfl-latest.zip" />
</wurfl>
```

用于 mode 特性的可能值有 Accuracy 和 Performance。或者，可以使用 InMemoryConfigurer 对象上的 SetMatchMode，以编程方式配置工作模式。

```
WURFLManagerBuilder.Build(
            new InMemoryConfigurer()
                .MainFile(...)
                .SetMatchMode(MatchMode.Accuracy));
```

这两种工作模式的区别是什么？区别只在于桌面用户代理。就移动设备来说——那些 is_wireless_device 性能设置为 true 的移动设备——这两种工作模式没什么区别，其整体的行为并无差异。而对于其他类型的设备来说，Performance 模式代表了一种快捷方式，能更快速地提供欠精确的响应。Performance 模式的影响是，如果用户代理是桌面浏览器，你就不会得到用于特定用户代理的有效性能值，只会得到与一般桌面浏览器相关的描述。

换句话说，Performance 模式只有在你需要迅速排除(或纳入)桌面浏览器而不去区分比如说 Internet Explorer 或 Chrome 或 Opera 的版本时才会成为一个选项。对于移动设备来说，Accuracy 或 Performance 通常提供一样的服务并返回最精确的响应。

请记住，可以全局指定工作模式(如前所述)，但也可以在每次调用的基础上指定，这可以通过使用 WURFL 管理器对象上 GetDeviceForRequest 方法的以下重载来实现：

```
public IDevice GetDeviceForRequest(WURFLRequest wurflRequest, MatchMode
     matchMode)
```

13.3.3　使用基于 WURFL 的显示模式

在本章前面我们讨论过，ASP.NET MVC 显示模式是路由控制器选取特定视图的理想方式。一般来说，显示模式不是特定于设备的模式，在某种意义上，你还可以使用显示模式直接切换到站点的灰度版本(译者注：灰度版本指的是正式版前的最后一个版本。)或者优化特定的浏览器，比如 Windows 8。

在任何情况下，使用显示模式，其路由到指定视图的逻辑总是取决于编码器的。如果打算构建用于多种设备的单个网站，只需要制定多个视图集——每个你支持的设备类别一个视图集。然后各设备类别便有了自己的显示模式。WURFL 有助于确保请求设备映射到适当的显示模式。这意味着平板电脑将获得指定页面的平板电脑视图(如果有的话)，而智能手机也将获得自己的视图。

重要提示：

这种方法如何确保你切实创建了多设备站点呢？在你有了特定于设备的首页后，基本就完成了。从特定于设备的首页中，可以指向所有视图通用的页面，也可以指向实现某种设备特定用例的页面。也可以决定使用一个特定于设备的控制器(比如 TabletXxxController)来处理特定于某种设备类别的用例，并将常规操作保持在你可能要使用的所有 XxxController 类中。

1. 选择显示模式

要有效地规划多设备站点，不管你打算使用什么样的实现技术和模式，第一步往往是对你要支持的设备类别有一个清晰的概念。如果希望只依赖 CSS，那么这将需要定义 RWD 断点(见第 12 章“让网站对移动端友好”)。或者，如果期望设置一个服务器端引擎并对请求设备提供特设标记，便需要定义显示模式。

在 global.asax 中，可以放置一个对某些 DisplayConfig 类的调用，这些类会定义和注册你网站支持的所有显示模式。

```
DisplayConfig.RegisterDisplayModes(DisplayModeProvider.Instance.Modes);
```

下面是用于 RegisterDisplayModes 方法的代码：

```
public class DisplayConfig
{
    public static void RegisterDisplayModes(IList<IDisplayMode> displayModes)
    {
        var modeDesktop = new DefaultDisplayMode("")
        {
            ContextCondition = (c => c.Request.IsDesktop())
        };
        var modeSmartphone = new DefaultDisplayMode("smartphone")
        {
            ContextCondition = (c => c.Request.IsSmartphone())
        };
        var modeTablet = new DefaultDisplayMode("tablet")
        {
            ContextCondition = (c => c.Request.IsTablet())
        };
        var modeLegacy = new DefaultDisplayMode("legacy")
        {
            ContextCondition = (c => c.Request.IsLegacy())
        };

        displayModes.Clear();
        displayModes.Add(modeSmartphone);
```

```
            displayModes.Add(modeTablet);
            displayModes.Add(modeLegacy);
            displayModes.Add(modeDesktop);
        }
    }
```

该方法首先会清除所有的默认模式，再添加四个模式：智能手机、平板电脑、桌面浏览器和传统手机。这意味着用来访问多设备站点的任何设备都将映射到这些模式中对应的那个。

在前面的代码中，IsLegacy、IsTablet 和 HTTP Request 对象上的其他方法都是普通的.NET 扩展方法，它们会接收 HTTP 上下文并针对“该请求是来自指定类型的设备吗？”问题返回一个布尔值答案。

2. 定义匹配规则

除了量身定做指定视图的内容外，你还可以有效地使用 WURFL 性能来确定某用户代理是否属于某特定类型的设备。你在 RegisterDisplayModes 方法的源代码中所看到的扩展方法通过使用 WURFL 性能实现了用户代理字符串的匹配规则。下面是一些可以用来检测智能手机的代码：

```
public static Boolean IsSmartphone(this HttpRequestBase request)
{
    return IsSmartPhoneInternal(request.UserAgent);
}
private static Boolean IsSmartPhoneInternal(String userAgent)
{
    var device = WURFLManagerBuilder.Instance.
                    GetDeviceForRequest(userAgent);
    return device.IsWireless() && !device.IsTablet() &&
           device.IsTouch() &&
           device.Width() > 240 &&
           (device.HasOs("android", new Version(2, 1)) ||
           device.HasOs("iphone os", new Version(3, 2)) ||
           device.HasOs("windows phone os", new Version(7, 1)) ||
           device.HasOs("rim os", new Version(6, 0)));
}
```

IsTouch、Width、HasOs 和上面代码中的其他方法就是其自身在 WURFL IDevice 接口上定义的扩展方法。下面是其代码:

```
public static class DeviceExtensions
{
    public static Boolean IsWireless(this IDevice device)
    {
```

```
        return device.GetCapability("is_wireless_device").ToBool();
    }

    public static Boolean IsTablet(this IDevice device)
    {
        return device.GetCapability("is_tablet").ToBool();
    }

    public static Boolean IsTouch(this IDevice device)
    {
        return device.GetCapability("pointing_method").Equals("touchscreen");
    }

    public static Int32 Width(this IDevice device)
    {
        return device.GetCapability("resolution_width").ToInt();
    }

    public static Boolean HasOs(this IDevice device, String os, Version version)
    {
        // Check OS
        var deviceOs = device.GetCapability("device_os");
        if (!deviceOs.Equals(os, StringComparison.
              InvariantCultureIgnoreCase))
            return false;

        // Check OS version
        var deviceOsVersion = device.GetCapability("device_os_version");
        if (!deviceOsVersion.Contains("."))
            deviceOsVersion = String.Format("{0}.0", deviceOsVersion);

        Version detectedVersion;
        var success = Version.TryParse(deviceOsVersion, out detectedVersion);
        if (!success)
            return false;

       return detectedVersion.CompareTo(version) >= 0;
    }
}
```

同样，ToInt 和 ToBool 也是将字符串解析成数字或布尔值的实用扩展方法。我在示例中使用扩展方法主要是为了清晰。当然，不用这些扩展方法你也能够达成同样的目的。

3. 运行中的多设备站点

显示模式的实际效果是，举个例子，当请求来自于平板电脑时，视图引擎子系统就会路由控制器以选取视图的平板电脑版本，如果有的话。鉴于之前的配置，所选的视图就符合名为 xxx.tablet.cshtml 的 Razor 文件，如图 13-4 所示。

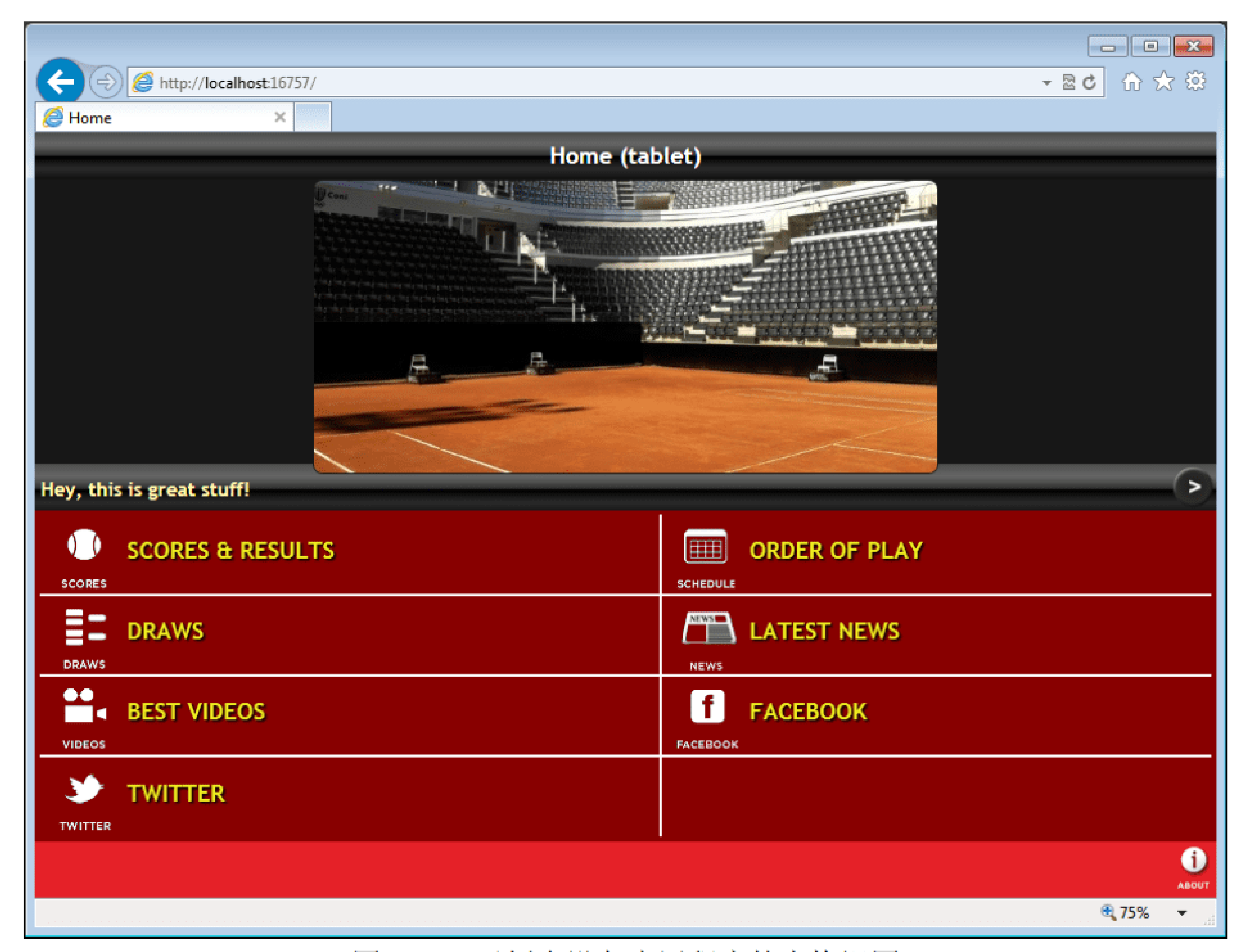
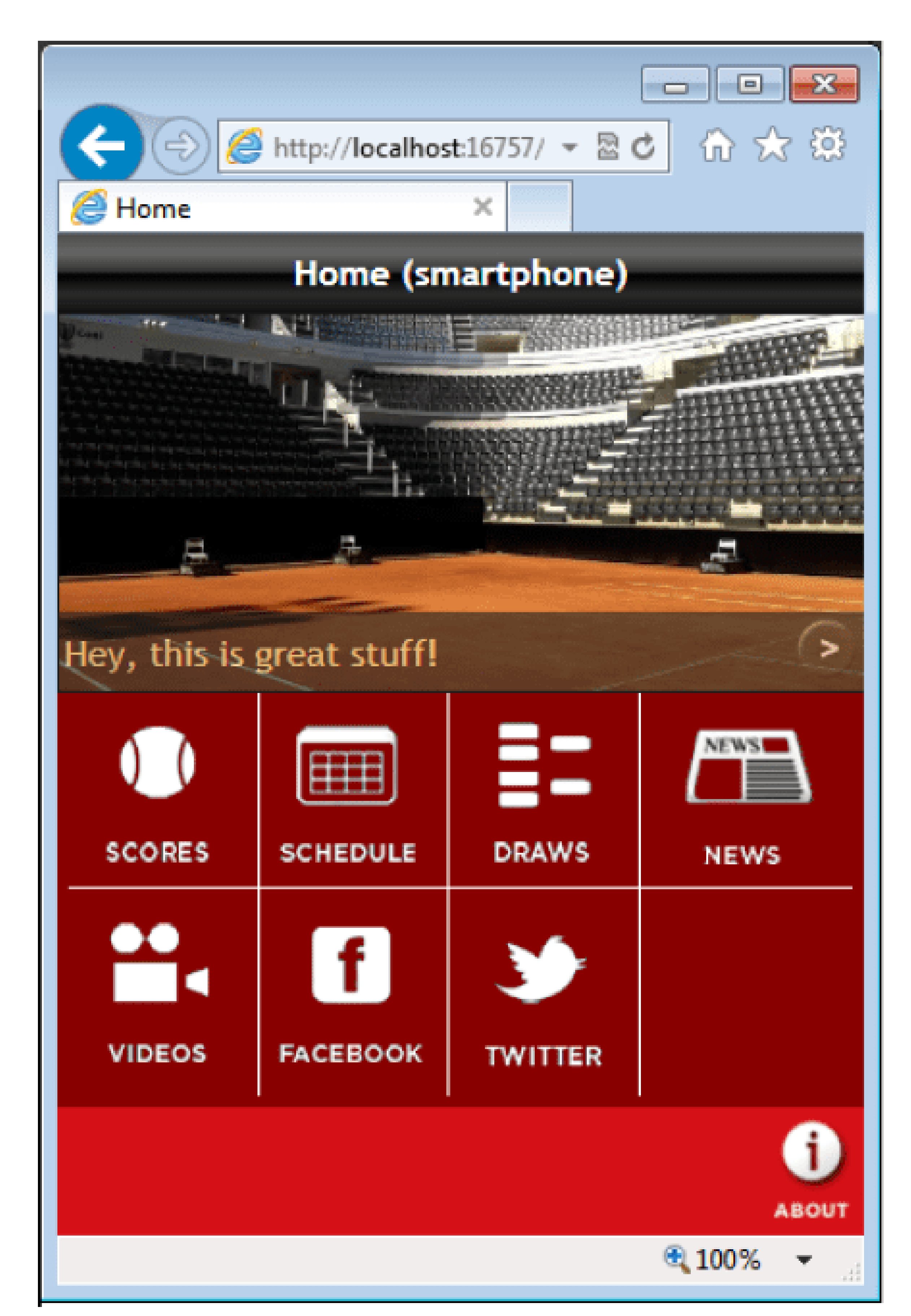


图 13-4　示例多设备应用程序的表格视图

相同的页面在智能手机和传统设备上看起来是不一样的，如图 13-5 所示。

你要做的就是使用多个 Razor 文件，为每个你希望不同于默认视图的视图分配一个。默认视图通常(但不一定)是桌面视图。要记住，显示模式是基于首个匹配项来选择的；因此你添加显示模式的顺序决定了你实际会拥有的视图回退机制。并不强制要求为应用程序的每个视图都准备一个智能手机文件或平板电脑文件；例如，如果桌面视图用在平板电脑上不产生任何问题，你就不需要有 xxx.tablet.cshtml 文件。



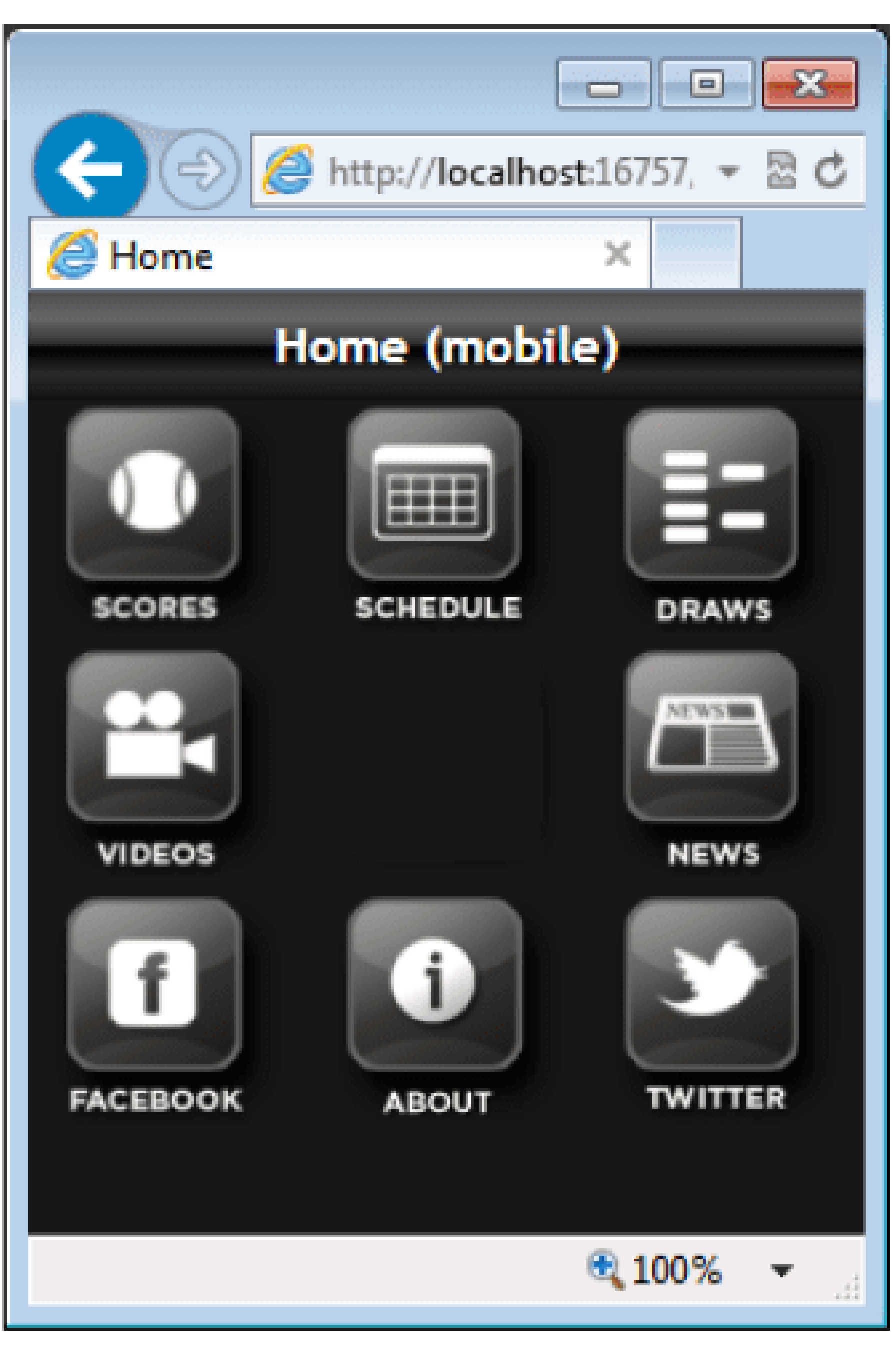


图 13-5　示例多设备应用程序的智能手机和传统手机视图

13.3.4　WURFL 云 API

在刚才讨论的示例中，我们使用了 WURFL 的内部部署版本，这需要你自己管理数据库，确保你总是具有最新的存储库，并且在出现问题时可以为数据库打上合适的补丁。不过，WURFL 还提供了一个云版本。

不足为奇的是，WURFL 云提供了多种服务方案，其范围涵盖了从免费但受限的方案到包月付费的几乎不受限制的访问选择。可以在 http://www.scientiamobile.com/cloud 找到详细内容。免费方案被限制为仅可选择两个性能，一个是 IP 地址，另一个是域，且每月不能超过 5000 次检测。

1. 设置 API

在订购其云服务之后，你便会拥有访问管理员面板和定义感兴趣性能的凭据。这时，可以使用类似于下面这样的代码：

```
var config = new DefaultCloudClientConfig
{
    ApiKey = "267026:ZpXrnoY3JOhfzMBd7CEyRS2acuq0H6NU"
};
var manager = new CloudClientManager(config);
```

```
var info = manager.GetDeviceInfo(httpContext, new[] { "is_wireless_device",
           "is_tablet" });
```

GetDeviceInfo 方法使用了 ASP.NET 请求的 HTTP 上下文和具有你希望接收的性能属性的一个数组。出于性能的原因，你请求的性能数量也可以比允许你请求的全部性能数量要小。

返回给你的对象包含一个名为 Capabilities 的字典，它带有所请求性能的所有值。更多详情，请访问 http://www.scientiamobile.com/wurflCloud/gettingStarted。

2. 云 API 与内部部署 API 的对比

面临使用 WURFL(一般是使用 DDR)时，选择云端比选择将所有一切存储在内部部署上更好吗？你需要做的权衡是非常容易理解的。

一方面，你具有单个 WURFL 请求的良好性能，如果它发生在你的 Web 服务器内部，明显会快很多。但是，另一方面，你需要付出购置和维护成本。内部部署许可的成本和分期摊销的速度都是要考虑的变量。另一个变量是定期更新 WURFL 数据库的成本，包括下载每周更新、重启应用程序和检查补丁文件的成本。最后，在内部部署方案中，你要负责处理任何可能追溯到的与 WURFL 数据库有关的扩展性问题。

在云方案中，你只要为可能所需的最小服务付费即可。但是，每个请求都要受到服务条款的规范，而且通常请求的处理都比较慢。最后一点是，你无法控制云端的数据库及其更新周期。

按照惯例，最佳选择往往要视具体情况而定。

13.4　为什么应该考虑服务器端解决方案

在 Web 开发中，可以实现多视角——即打算在不同设备上呈现的内容，这些设备包括平板电脑、笔记本电脑和智能手机——可以用后面两种方式中的一种来实现。可以使用单独的一组页面和多个辅助资源，比如 CSS、图片和脚本文件，或者可以使用多组页面和相关的资源。在这两种情形中，站点的后端都是一样的，业务逻辑也几乎是相同的。

但你想要的是“几乎一样”还是“完全一样”呢？

这是可能影响你决策过程的关键点之一。如果希望在平板电脑、智能手机和笔记本电脑上的体验必须相同，那么 RWD 和客户端解决方案是最理想的办法。RWD 技术是创建单个网站的技术，它能根据显示屏幕的大小自动适应和调整呈现效果，无论屏幕是属于智能手机或平板电脑的，还是属于桌面计算机或智能电视的。检测屏幕大小和应用正确样式表的工作是由 Web 浏览器执行的。

这听起来是不是像完美的解决方案？你是应该就此止步还是再多考虑一下呢？让我们看看再一次的思量还会带来什么。

RWD 被广为诟病的是，它不会真正地区分设备，而是只区分屏幕大小。这样的后果就是，它可能会向连接到很慢的 3 G 网络的小设备发送很多的内容。换句话说，RWD 是处理多个屏幕尺寸的有效方法，但不一定是处理带有不同特征的多种设备的有效方法，这些特征也包括不同的屏幕尺寸。RWD 中你很难处理的一种情况是，你需要为通过特定设备连接的用户提供不同的功能、布局和用例，但要如何才能达成目的呢？

简而言之，要做出明智决定，必须回答的关键问题是，你是否想要提供一种独特的、特定于设备的体验。你想要为平板电脑、智能手机和笔记本电脑做不同的分析和设计吗？一个设计对你来说足够吗？如果一个设计就能满足需求，那就请继续使用 RWD 并使用实现技巧来减少下载和性能问题。如果一个设计不能让你满意，那么请专注于用户代理的服务器端分析的技术和产品，以识别请求设备的类别。

当可以使用单个设计/项目来适应任何设备的时候，客户端解决方案就能够胜任了。当你希望在单个网站的上下文中拥有一个用于每个设备类别的特定设计/项目时，服务器解决方案是比较合适的。

13.5　本章小结

并没有太多的网站真的需要求助于服务器端设备检测。对于大多数站点来说，一个基于 RWD 原则的客户端解决方案就已经足够了。但是，这并不意味着客户端解决方案总是优于服务器端解决方案。

服务器端解决方案本质上比单纯基于 RWD 的客户端解决方案要更具灵活性。

要克服 RWD 解决方案的一些结构性限制，你需要添加一些服务器端逻辑。这意味着你要识别请求设备，搞清楚它的性能，然后提供针对性的标记。这可以保证一个 800 像素的平板电脑将会接收为移动用户量身定做的内容，而通过 800 像素浏览器窗口连接的用户将接收到特定于桌面端的内容。使用客户端逻辑，你不能确定查看页面的浏览器是托管在移动设备上还是托管在小尺寸的浏览器窗口上。

本章提供了一个服务器端检测的有力观点；请使用它来匹配设备类别(智能手机、平板电脑、智能电视、笔记本电脑以及其他任何你能想到的设备)并为这些设备提供你心目中的站点。ASP.NET MVC 的显示模式特征使你能够很容易地创建出带有你可能需要的多种外观(和功能组)的单个网站。